Sandboxed Application

Developer's Guide

MicroEJ 4.1

Reference:	TLT-0788-DGI-SandboxedApplicationDeveloperGuide-MicroEJ
Version:	4.1
Revision:	B

Revision History		
Revision 4.1-B	05/2017	
Alignment with online Getting Started. Removed board specific content. Updated some schematics.		
Revision 4.1-A	04/2017	
Updates for MicroEJ 4.1, new schematics and new screenshots according to firmwares and virtual devices naming convention. Added a JUnit testsuite section and Wadapps Administration Console tool section.		
Revision 4.0-C	07/2016	
Added instructions related to the required differentiation of Application-Id fields in MANIFEST.MF files for both Shared Interface provider and user applications		
Revision 4.0-B	07/2016	
MicroEJ Classpath chapter review.		
Revision 4.0-A	06/2016	
Initial release.		

Table of Contents

List of Figures

List of Tables

Chapter 1. MicroEJ Overview

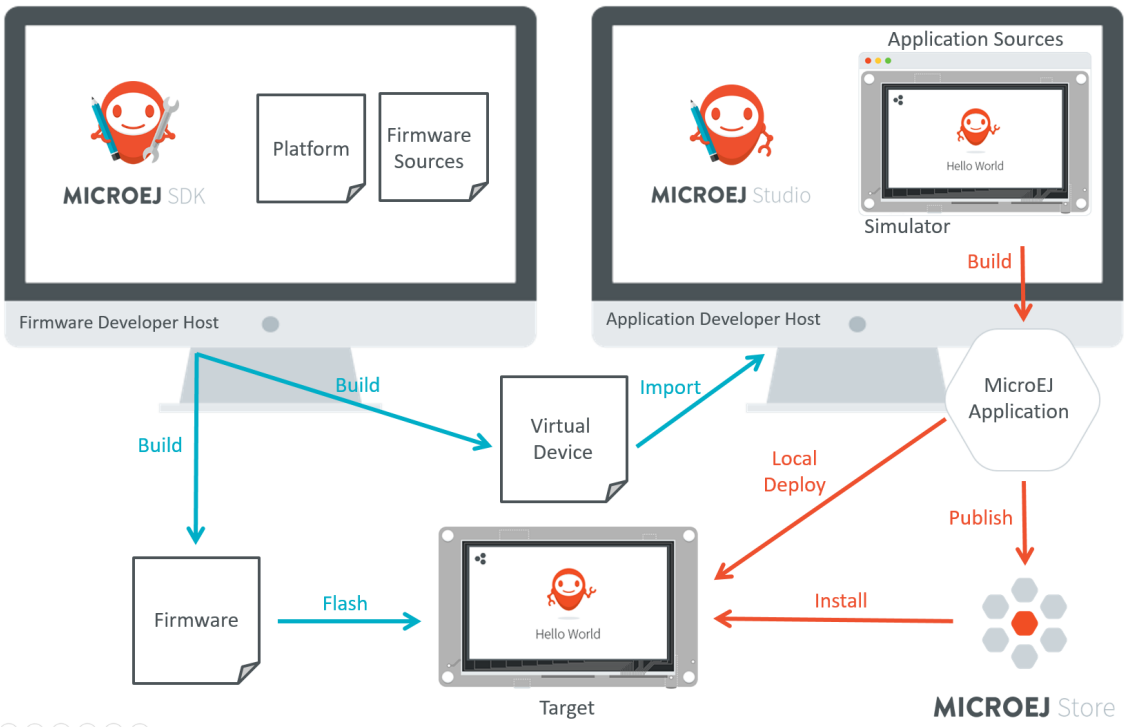
1.1. MicroEJ Editions

MicroEJ offers a comprehensive toolset to build the embedded software of a device. The toolset covers two levels in device software development:

- MicroEJ SDK for device firmware development
- MicroEJ Studio for application development

The firmware will generally be produced by the device OEM, it includes all device drivers and a specific set of MicroEJ functionalities useful for application developers targeting this device.

Figure 1.1. MicroEJ Development Tools Overview



Using the MicroEJ SDK tool, a firmware developer will produce two versions of the MicroEJ binary, each one able to run applications created with the MicroEJ Studio tool:

- A firmware binary to be flashed on OEM devices
- A Virtual Device which will be used as a device simulator by application developers

Using the MicroEJ Studio tool, an application developer will be able to:

- Import Virtual Devices matching his target hardware in order to develop and test applications on the simulator.
- Deploy the application locally on an hardware device equipped with the MicroEJ firmware

- Package and publish the application on a store, enabling remote end users to install it on their devices.

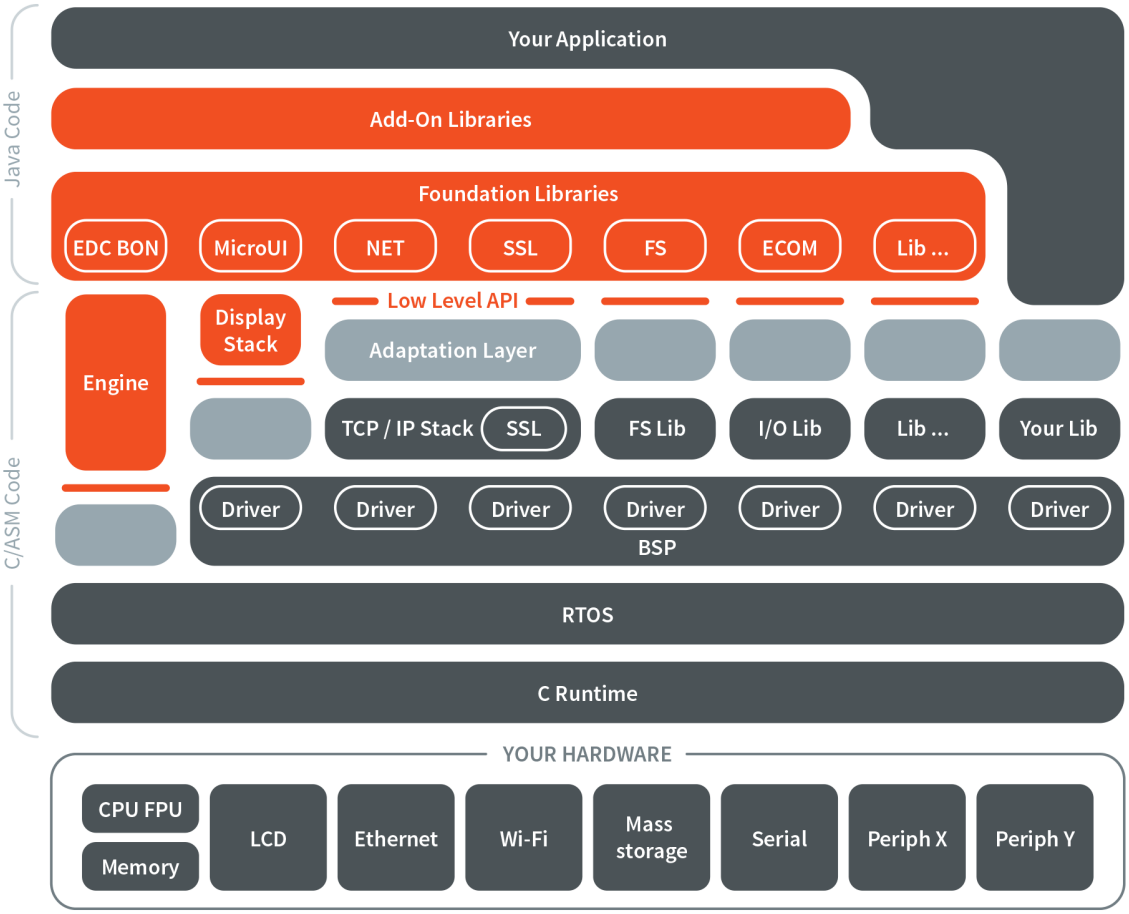
1.2. Firmware

1.2.1. Bootable Binary with Core Services

A MicroEJ firmware is a binary software program that can be programmed into the flash memory of a device. A MicroEJ firmware includes an instance of a MicroEJ runtime linked to:

- underlying native libraries and BSP + RTOS,
- MicroEJ libraries and application code (C and Java code).

Figure 1.2. MicroEJ Firmware Architecture



1.2.2. Specification

The set of libraries included in the firmware and its dimensioning limitations (maximum number of simultaneous threads, open connections, ...) are firmware specific. Please refer to `http://developer.microej.com/getting-started.html` firmware release notes.

1.3. Virtual Device

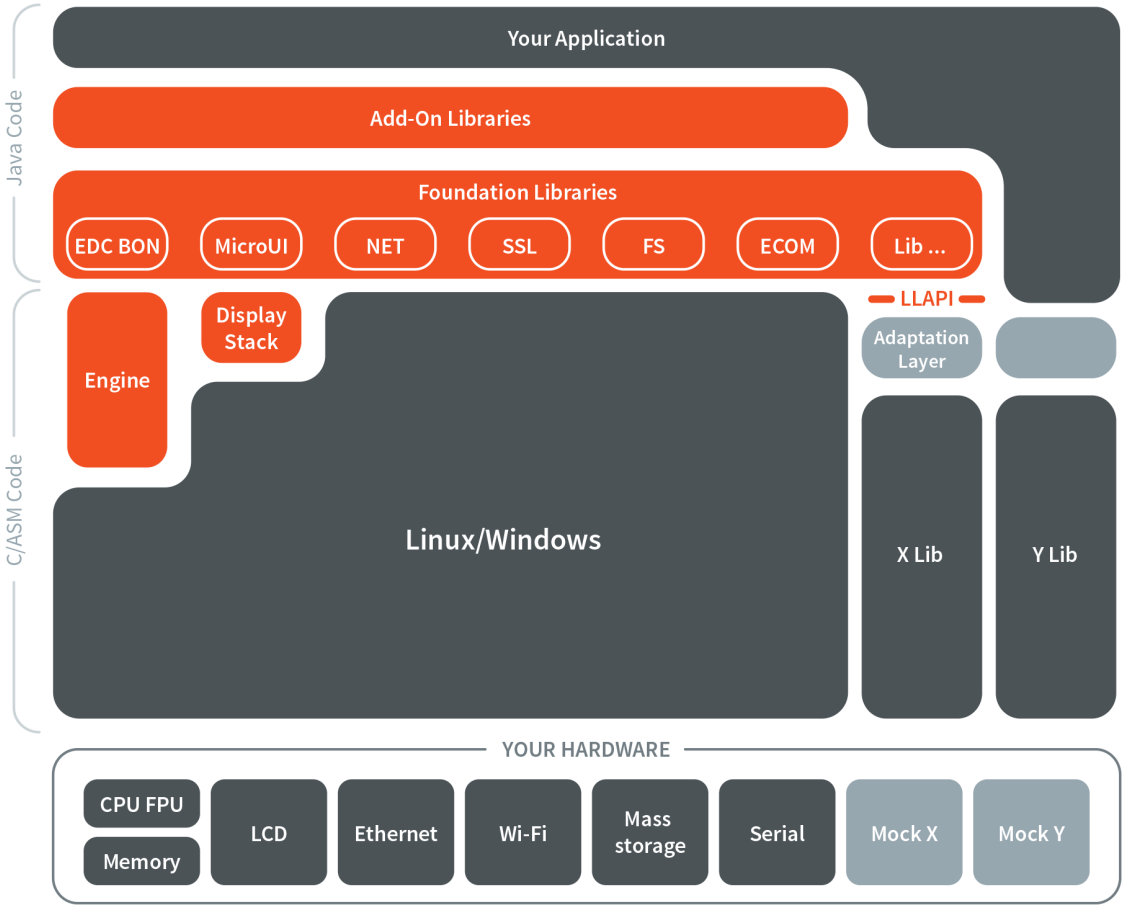
1.3.1. Using a Virtual Device for Simulation

The virtual device includes the same custom MicroEJ Core, libraries and resident applications as the real device. The virtual device allows developers to run their applications either on the Simulator, or directly on the real device through local deployment.

The Simulator runs a mockup board support package (BSP Mock) that mimics the hardware functionality. An application on the simulator is run as a standalone application

Before an application is locally deployed on device, MicroEJ Studio ensures that it does not depend on any API that is unavailable on the device.

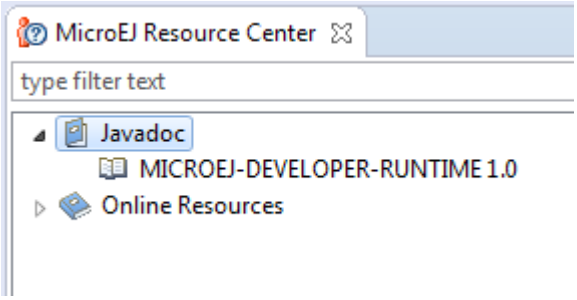
Figure 1.3. MicroEJ Virtual Device Architecture



1.3.2. Runtime Environment

The set of MicroEJ APIs exposed by a virtual device (and therefore provided by its associated firwmare) is documented in Javadoc® format in the MicroEJ Resource Center (`Window > Show View > MicroEJ Resource Center`).

Figure 1.4. MicroEJ Resource Center APIs



Chapter 2. MicroEJ Studio Getting Started

2.1. Introducing MicroEJ Studio and Virtual Devices

MicroEJ Studio provides tools based on Eclipse to develop software applications for MicroEJ-ready devices. MicroEJ Studio allows application developers to write MicroEJ applications, run them on a virtual (simulated) or real device, and publish them to the MicroEJ Application Store.

This document is an introduction to application development with MicroEJ Studio. The purpose of MicroEJ Studio is to develop for targeted MCU/MPU computers (IoT, wearable, etc.) and it is therefore a cross-development tool.

Unlike standard low-level cross-development tools, MicroEJ Studio offers unique services like hardware simulation, local deployment to the target hardware and final publication to a MicroEJ Application Store.

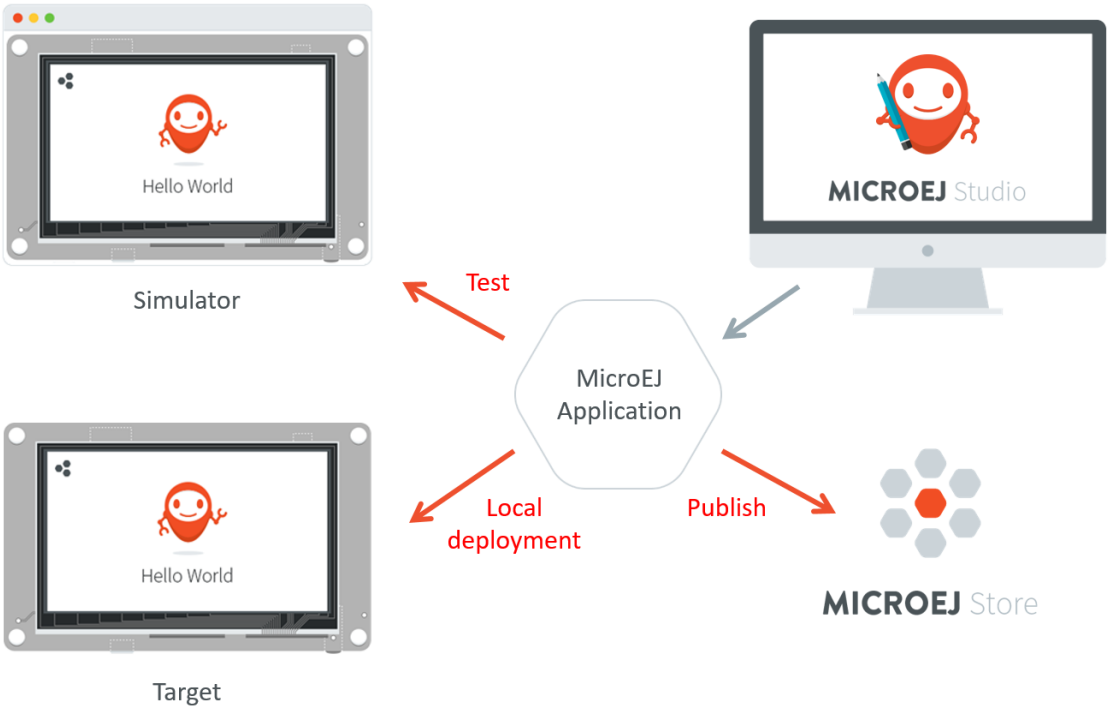
Application development is based on the following elements:

- MicroEJ Studio, the integrated development environment for writing applications. It is based on Eclipse and is relies on the integrated Java compiler (JDT). It also provides a dependency manager for managing MicroEJ Libraries (see Section 8.5, “Library Dependency Manager”). The current version of MicroEJ Studio is built on top of Eclipse Mars (`http://www.eclipse.org/downloads/packages/release/Mars/2`).
- MicroEJ Virtual Device, a software package including the resources and tools required for building and testing an application for a specific MicroEJ-ready device. A Virtual Device will simulate all capabilities of the corresponding hardware board:
  - Computation and Memory
  - Communication channels (e.g. Network, USB ...)
  - Display
  - User interaction

  Virtual Devices are imported into MicroEJ Studio within a local folder called MicroEJ Platforms repository. Once a Virtual Device is imported, an application can be launched and tested on simulator. It also provides a means to locally deploy the application on a MicroEJ-ready device.
- MicroEJ-ready device, an hardware device that has been previously programmed with a MicroEJ firmware. A MicroEJ firmware is a binary instance of MicroEJ runtime for a target hardware board. MicroEJ-ready devices are built using MicroEJ SDK. MicroEJ Virtual Devices and MicroEJ Firmwares share the same version (there is a 1:1 mapping).

The following figure gives an overview of MicroEJ Studio possibilities:

Figure 2.1. MicroEJ Application Development Overview



2.2. Perform Online Getting Started

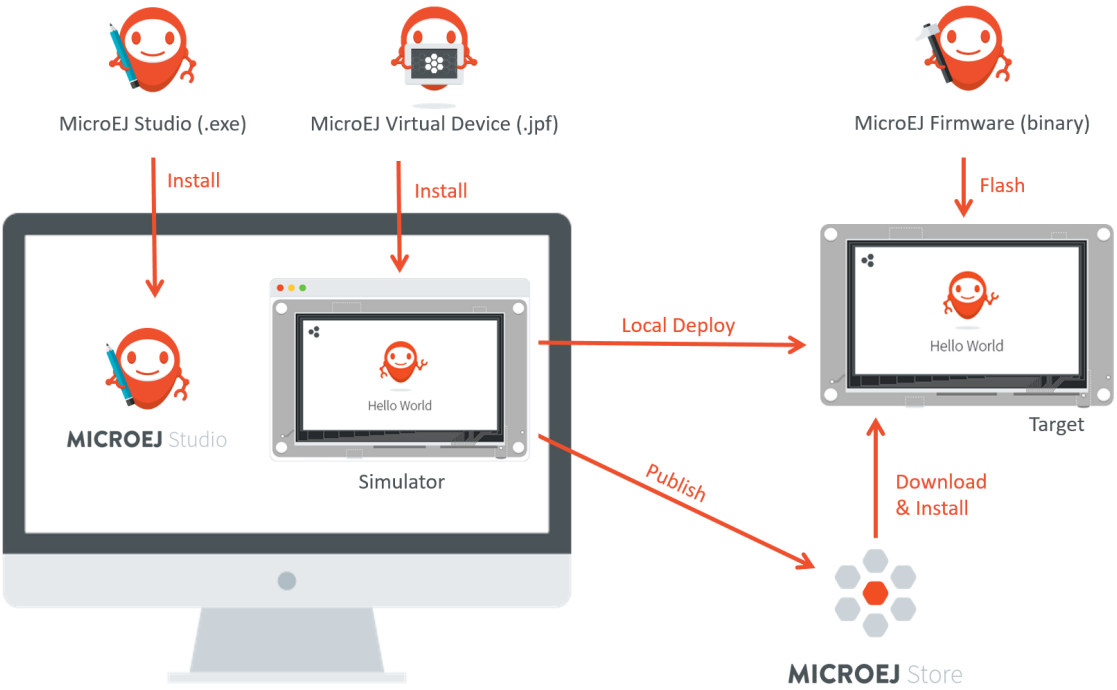
MicroEJ Studio Getting Started is available on `http://developer.microej.com/getting-started.html`.

Starting from scratch, the steps to go through the whole process are:

1. Setup a board and test a MicroEJ firmware:
    - Select between one of the available boards
    - Download and install a MicroEJ firmware on the target hardware
    - Deploy and run a MicroEJ demo on board
2. Setup and learn to use development tools:
    - Download and install MicroEJ Studio
    - Download and install the corresponding Virtual Device for the target hardware
    - Download, build and run your first application on simulator
    - Build and run your first application on target hardware

The following figure gives an overview of the MicroEJ software components required for both host computer and target hardware:

Figure 2.2. MicroEJ Studio Development Imported Elements

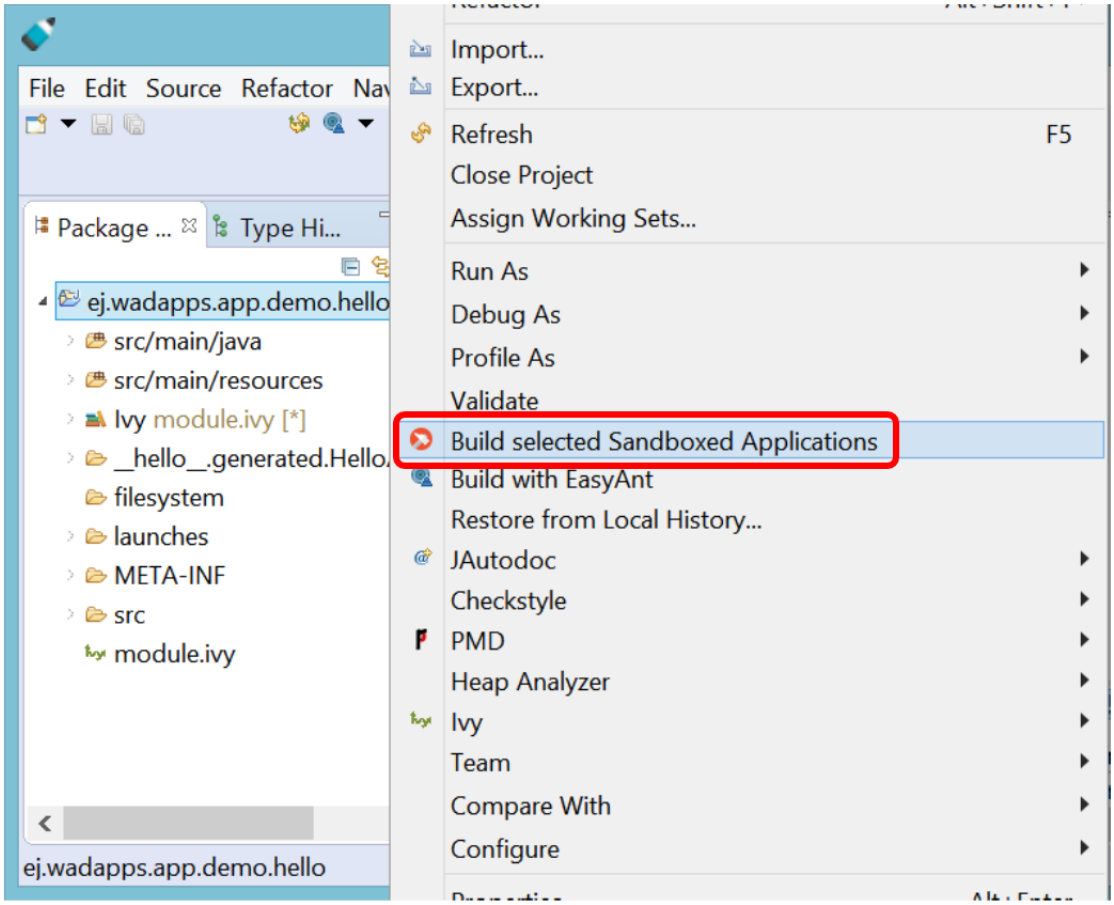


2.3. Application Publication

2.3.1. Build the WPK

When the application is ready for deployment, the last step in MicroEJ Studio is to create the WPK (Wadapps PacKage) file that is intended to be published on a MicroEJ Store for end users.

In MicroEJ Studio, right click on project name and choose: `Build Selected Sandboxed Applications`.

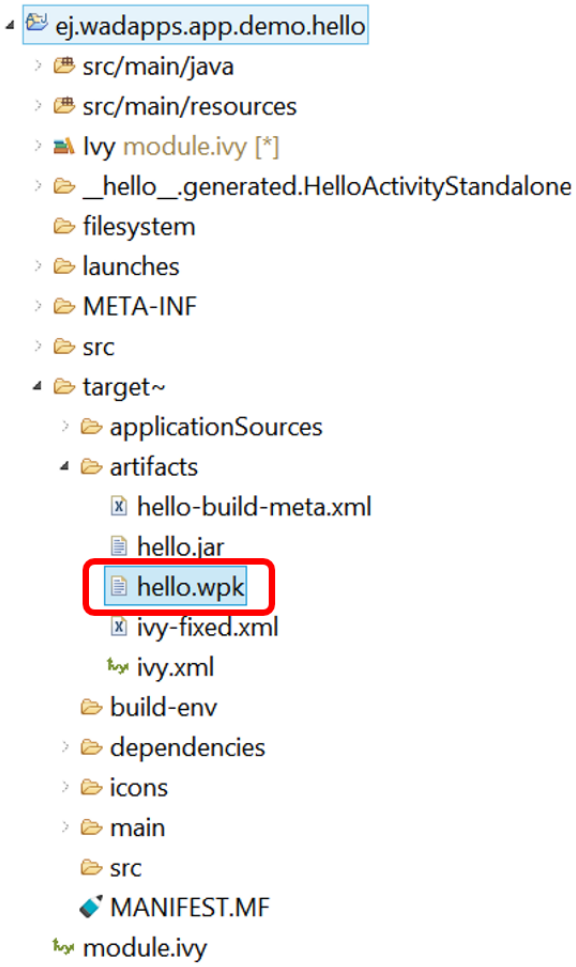


The WPK build process will display messages in MicroEJ console, ending up with a `BUILD SUCCESSFUL` message.

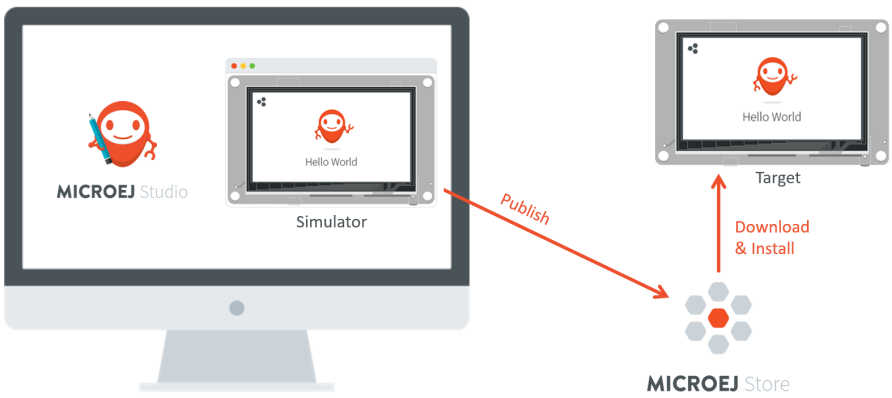
```
    [echo] project hello published locally with version 1.0.0-RC201605111118

BUILD SUCCESSFUL

Total time: 4 seconds
---- Memory Details ----
  Used Memory  = 66MB
  Free Memory  = 80MB
  Total Memory = 146MB
-----------------------
```

2.3.2. Publish on a MicroEJ Store

The WPK file produced by the build process is located in a dedicated `target~/artifacts` folder in the project.



The `.wpk` file is ready to be uploaded to a MicroEJ Store. Please consult `https://community.microej.com` for more information.



2.4. Workspaces and Platforms Repositories

When starting MicroEJ Studio, it prompts you to select the last used workspace or a default workspace on the first run. A workspace is a main folder where to find a set of projects containing source code.

When loading a new workspace, MicroEJ Studio prompts for the location of the MicroEJ Platforms repository. By default, MicroEJ Studio suggests to point to the default MicroEJ Platforms repository on

your operating system, located at `${user.home}/.microej/repositories/[version]`). You can select an alternative location. Another common practice is to define a local repository relative to the workspace, so that the workspace is self-contained, without external file system links and can be shared within a zip file.

2.5. Application Development

The following sections of this document shall prove useful as a reference when developing applications for MicroEJ. They cover concepts essential to MicroEJ applications design.

In addition to these sections, by going to `http://developer.microej.com/`, you can access a number of helpful resources such as:

- Libraries,
- Application Examples, with their source code,
- Documentation (HOWTOs, Reference Manuals, APIs javadoc...)

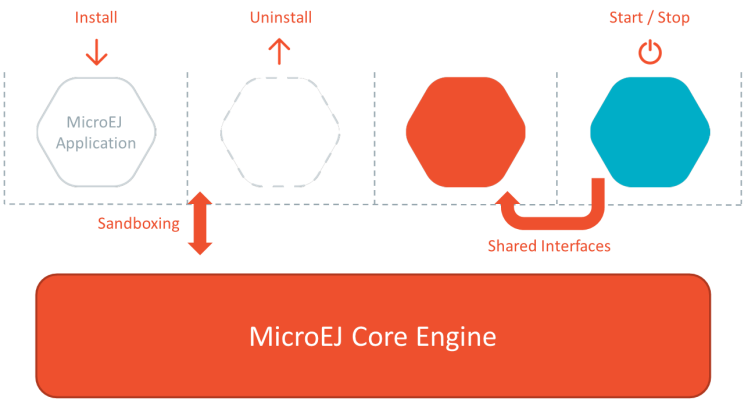
Chapter 3. Wadapps Framework

3.1. MicroEJ Component Framework

MicroEJ offers a multi-application execution framework called Wadapps framework. The basic features offered by the Wadapps framework for each application include:

- Dynamic installation and uninstallation
- Execution lifecycle management (Activities and Background Services)
- Services usage
- Inter-application communication (Chapter 7, *Shared Interfaces*)

Figure 3.1. Wadapps Framework Components View



3.2. Execution Lifecycle

Depending on the application nature, two execution modes are available in the Wadapps framework:

- Background Service
- Activity

Background Service is suitable for applications with no graphic interface, whereas Activity is dedicated to applications using the screen and user interface. An application must declare at least one background service or activity, and can declare a mix of both.

3.2.1. Background Service Lifecycle

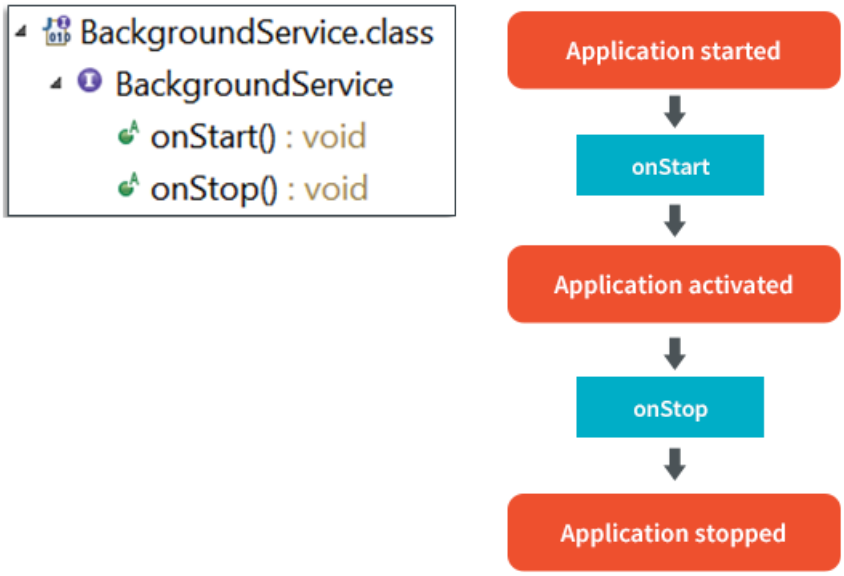
A background service entry point is a class that extends the `ej.wadapps.app.BackgroundService` interface which offers a small set of methods dedicated to the lifecycle of an application with no graphic interface:

- `public void onStart()`

- `public void onStop()`

Usually, a background service has a unique active state. The `onStart()` method is called just after the application has been started and gives the entry point to start its job. This can be just starting a thread or simply registering a shared service (see Section 7.4, “System Registries”). The `onStop()` method is called just before the application is stopped and gives to the application the opportunity to properly save its state. Note that background service lifecycle methods are assumed to return quickly. In case of long blocking code, a new thread must be created.

Figure 3.2. Background Service Lifecycle within an application



3.2.2. Activity Lifecycle

An activity entry point is a class that extends the `ej.wadapps.app.Activity` interface which offers a more comprehensive set of methods dedicated to the lifecycle of an application with a graphic interface:

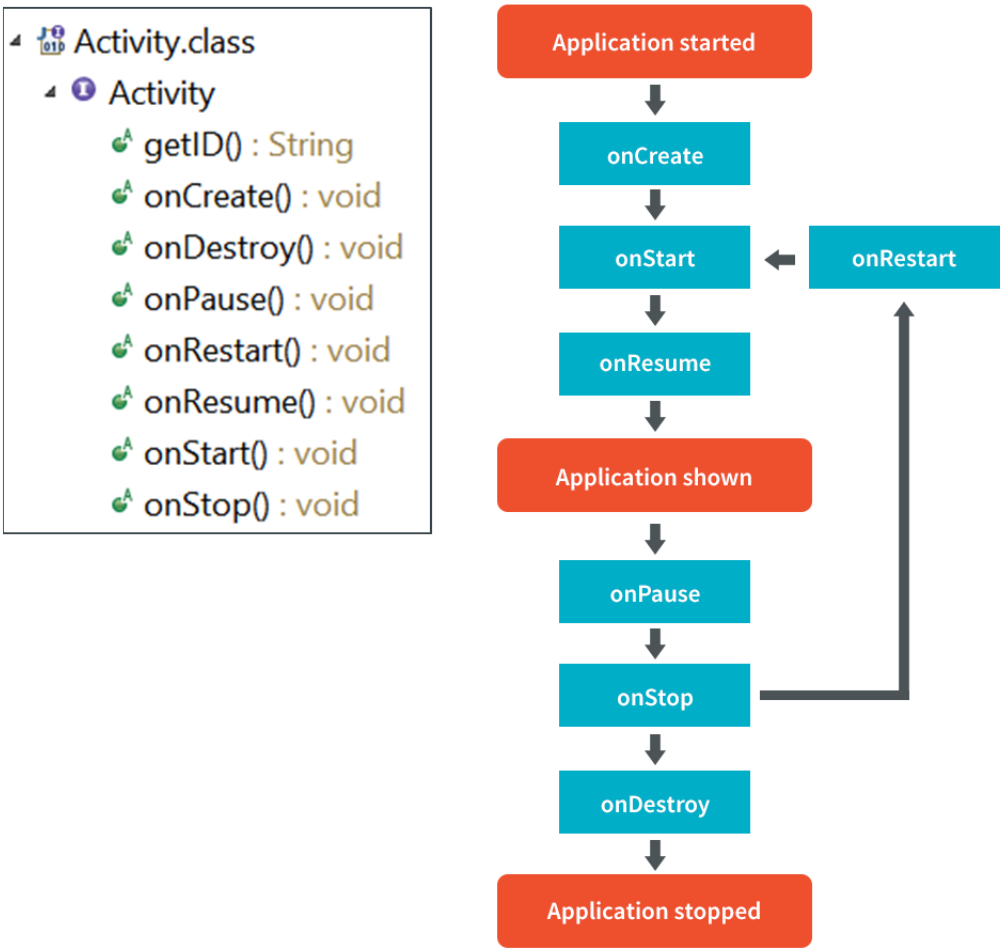
- `public void onCreate()`
- `public void onDestroy()`
- `public void onStart()`
- `public void onRestart()`
- `public void onStop()`
- `public void onPause()`
- `public void onResume()`

Note that as for a background service, activity lifecycle methods are assumed to return quickly. In case of long blocking code, a new thread must be created.

An activity must share the Graphical User Interface with other activities, either from the same application or from different ones. As a consequence the implementation of the Activity interface must handle transitions between several activity states:

- CREATED
- STARTED
- PAUSED

Figure 3.3. Activity Lifecycle Within an Application



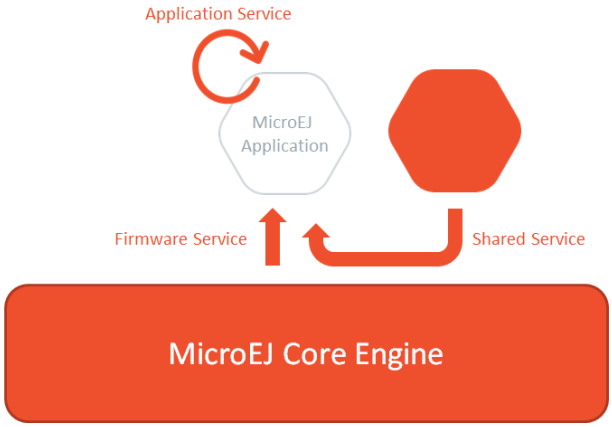
3.3. Services Usage

The Wadapps framework provides a service oriented mechanism where generic services may be provided on several levels:

- Application local implementation
- MicroEJ Firmware provided service
- Service shared by another application

Services retrieval order follows the order of the previous list. An application local implementation may override a MicroEJ Firmware provided service. A MicroEJ Firmware provided service cannot be overridden by a service shared by an other application.

Figure 3.4. Wadapps Services Providers



3.3.1. Retrieving Services

Services are retrieved in a transparent way however they have been published, using the default service loader. Given a class that represents the service API, it returns the registered implementation. The default service loader can be retrieved using `ej.components.dependencyinjection.ServiceLoaderFactory.getServiceLoader()`.

Then, the service implementation is retrieved using `ej.components.dependencyinjection.ServiceLoader.getService(Class)`.

Next figure is an example for retrieving the `ej.wadapps.storage.Storage` service.

Figure 3.5. Wadapps Service Retrieval Example

```
// 1- Retrieve the default ServiceLoader instance
ServiceLoader sl = ServiceLoaderFactory.getServiceLoader();
// 2- Retrieve the Storage service implementation
Storage storageService = sl.getService(Storage.class);
System.out.println("Implementation Name = "+
        storageService.getClass().getName());
```

3.3.2. Application Local Services

Application local services are provided as an application's local class and declared in the `META-INF/services` section of the project (see Section 4.3.4, “Services Folder”).

3.3.3. Shared Registry

External services may be provided by an application to another application through the Shared Registry mechanism (see Section 7.4, “System Registries”).

3.4. Standalone vs Sandboxed Application

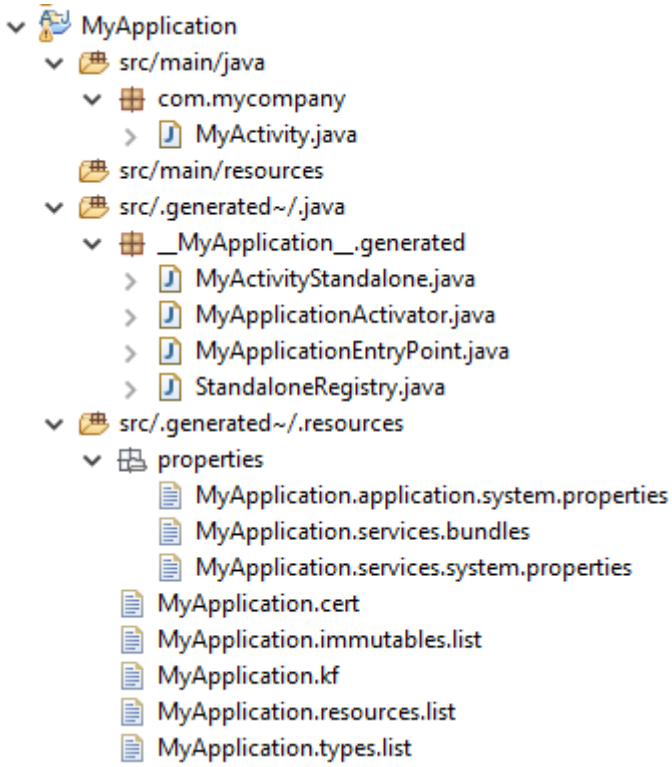
A standalone application is an application that defines a main entry point (a class that contains a `public static void main(String[])` method). A standalone application can be run on the simulator and is intended to be statically linked with a platform to produce a firmware.

A sandboxed application is an application that is defined in MicroEJ Studio with the sandboxed application structure (see Chapter 4, *Sandboxed Application Structure*). A sandboxed application is intended to be dynamically deployed on a firmware. MicroEJ Studio provides a bridge for using a sandboxed application as a standalone application by autogenerating standalone main entry points and allowing to fetch standalone specific dependencies.

3.4.1. Automatically Generated Standalone Entry Points

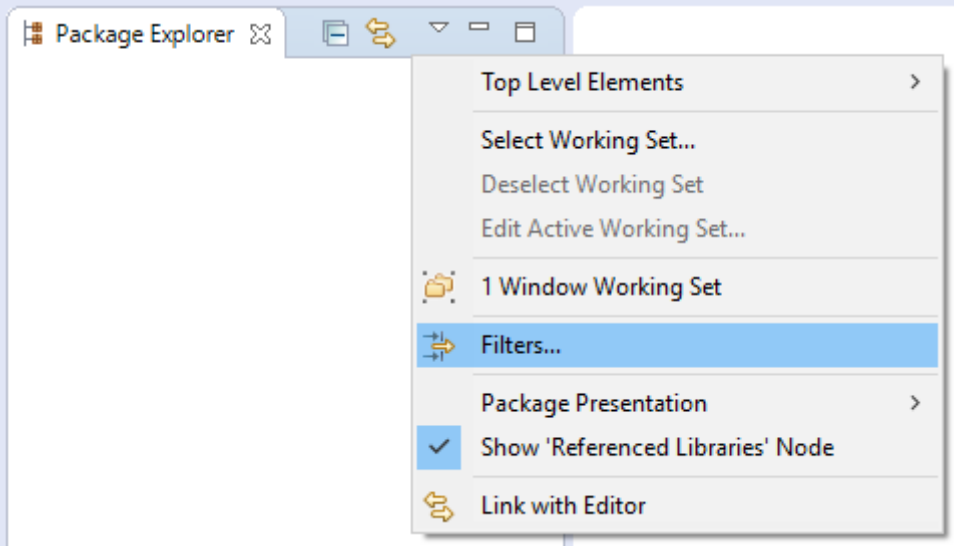
For a sandboxed application, MicroEJ Studio automatically generates standalone main entry point. The main entry point is in charge to start the wadapps framework that will activate declared Activities and BackgroundServices. One specific main entry point is generated per declared Activity. Standalone classes names have the `Standalone` suffix. The autogenerated code is located in `src/.generated~/java` source folder of a sandboxed application project.

Figure 3.6. Sandboxed Application Autogenerated Structure



The `src/.generated` folder is hidden by default. To make it visible, in `Packages Explorer` select the `Filters...` menu and check `.* resources` item.

Figure 3.7. Package Explorer **`Filters...`** Menu



3.4.2. Standalone Application Specific Dependencies

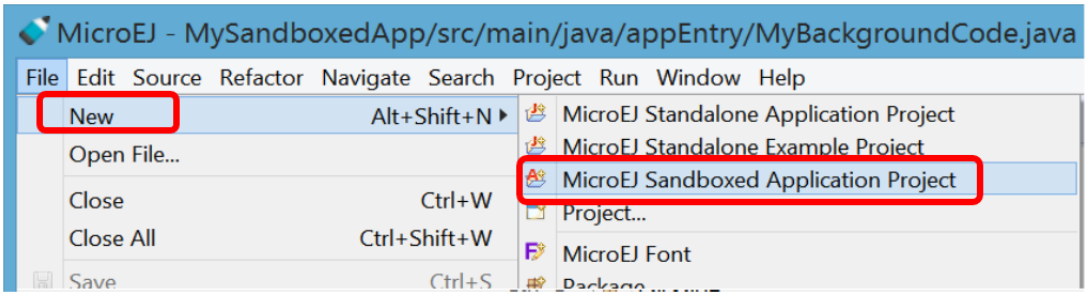
MicroEJ allows to declare additional dependencies that will be taken into account only when launching a standalone application such as the Simulator. This is done by defining a built-in Ivy configuration named `microej.launch.standalone` in the `module.ivy` file of a sandboxed application project. Refer to the `module.ivy` generated by the application template (Section 4.1, “Application Template Creation”) to get the list of required standalone libraries.

```
<dependencies>
     […]
 <!--
  A Classpath dependency only used by
  standalone application launches
 -->
 <dependency
  org="com.mycompany" name="xxx" rev="xxx"
  conf="microej.launch.standalone->*"
 />
</dependencies>
```

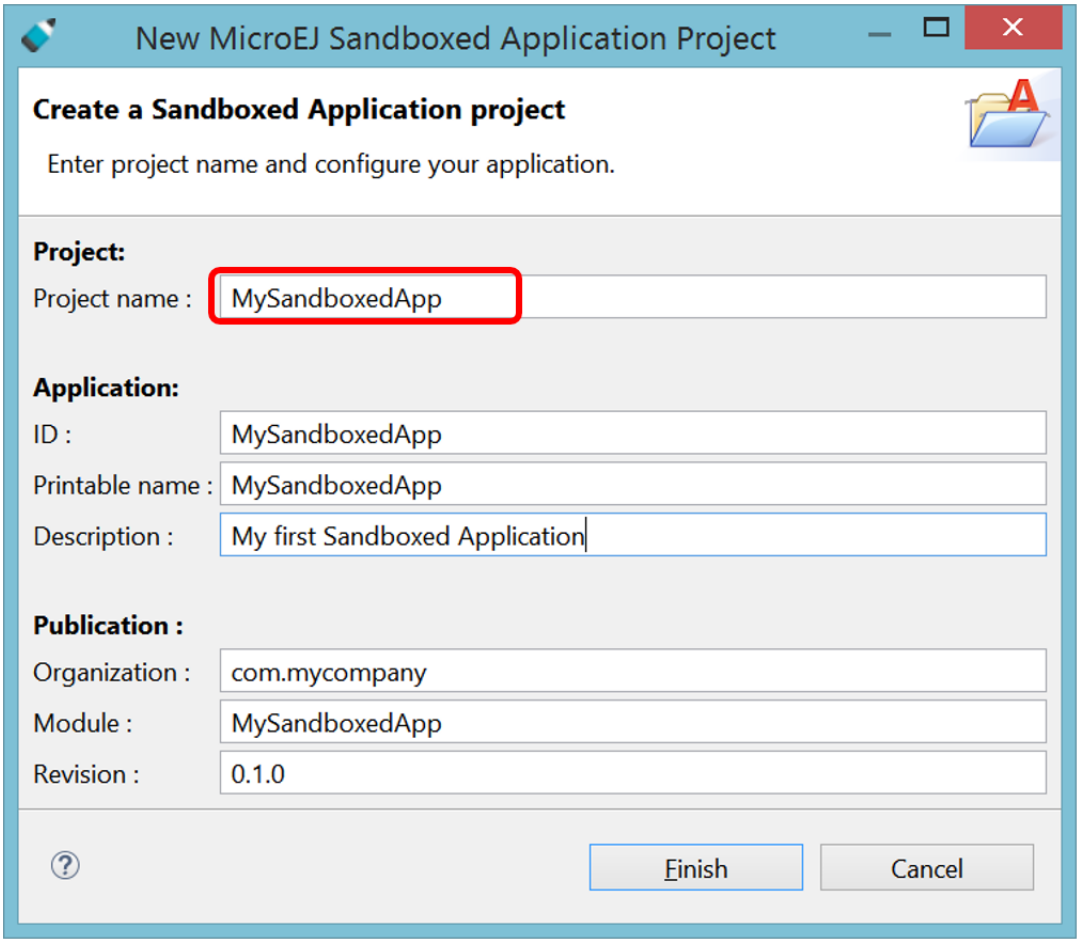
Chapter 4. Sandboxed Application Structure

4.1. Application Template Creation

The first step to explore a sandboxed application structure is to create a new projet for the development of a graphical application. First select `File > New > MicroEJ Sandboxed Application Project`:

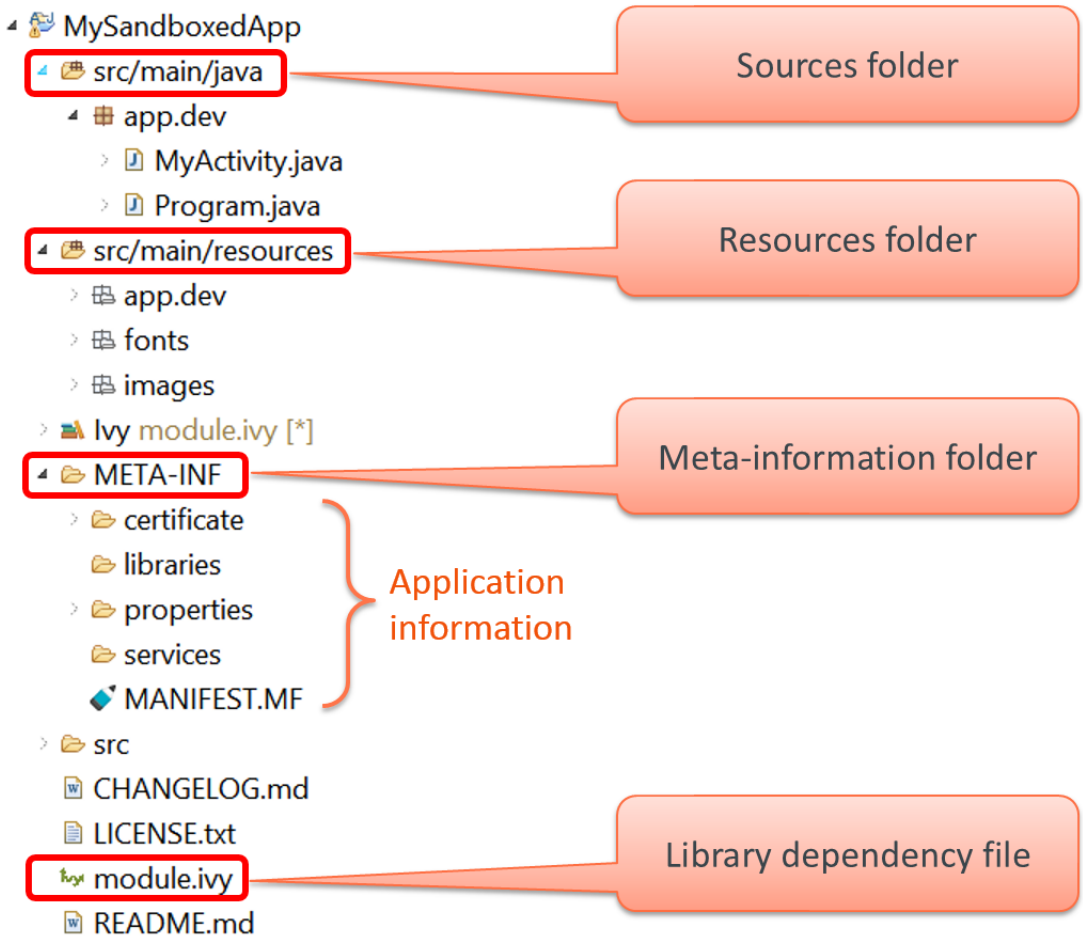


Fill in the application template fields, the `Project name` field will automatically duplicate in the following fields.



A template project is automatically created and ready to use, this project already contains all places where the application developer will put content:

- `src/main/java`: Folder for future sources
- `src/main/resources`: Folder for future resources (images, fonts etc.)
- `META-INF`: Sandboxed application configuration and resources
- `module.ivy`: Ivy input file, dependencies description for the current project



The Ivy section contains the list of dependencies automatically resolved by Ivy from the content of `module.ivy`, from a development perspective this section is read-only.

The application functionalities will determine which parts of this structure are impacted, for example the development of a simple "Hello world" application will only impact the `src/main/java folder` and `META-INF/MANIFEST.MF` file.

4.2. Sources Folder

The project source folder (`src`) contains two areas:

- Source

- Resources

Source folder will contain all `.java` files of the project, resources folder will contain elements that the application will use at runtime like raw resources, images or character fonts.

4.3. META-INF Folder

The `META-INF` folder contains several folders and one file named the manifest file described hereafter.

4.3.1. Certificate Folder

Contains certificate information used during the application deployment.

4.3.2. Libraries Folder

Contains a list of additional libraries useful to the application and not resolved through the regular transitive dependency check

4.3.3. Properties Folder

Contains an `application.properties` file which contains application specific properties that can be accessed at runtime.

4.3.4. Services Folder

Contains a list of files that describe local services provided by the application (see Section 3.3.2, “Application Local Services”). Each file name represents a service class fully qualified name, and each file contains the fully qualified name of the provided service implementation.

4.3.5. Manifest File

The file `META-INF/MANIFEST.MF` is initialized with the information given on project creation, extra information may be added to this file to declare the entry points of the application.

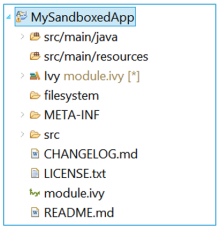
4.4. **`module.ivy`** File

The `module.ivy` file contains a description of all the libraries required by the appliction at runtime (see Section 8.5, “Library Dependency Manager”).

Chapter 5. Background Service Application

5.1. Create a Sandboxed Application Project

In MicroEJ menu, select: `File > New > MicroEJ Sandboxed Application Project` an give `MySandboxedApp` as the project name, a template project is automatically created and ready to use.



For the detailed content of the project structure, please consult section Chapter 4, *Sandboxed Application Structure*. Here is a list of the elements we will modify for the simple sandboxed application:

- `src/main/java`: Add source files
- `META-INF/MANIFEST.MF`: Set application's BackgroundService entry point

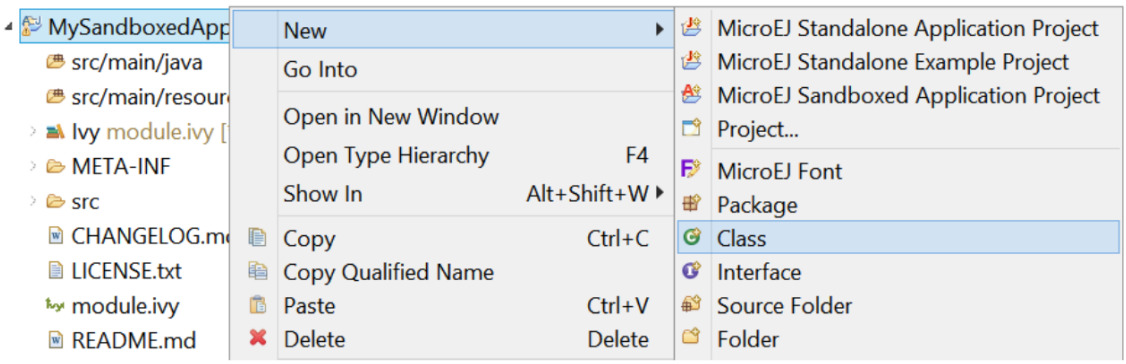
5.2. Fill the Application Structure

5.2.1. Simple Background Application Code

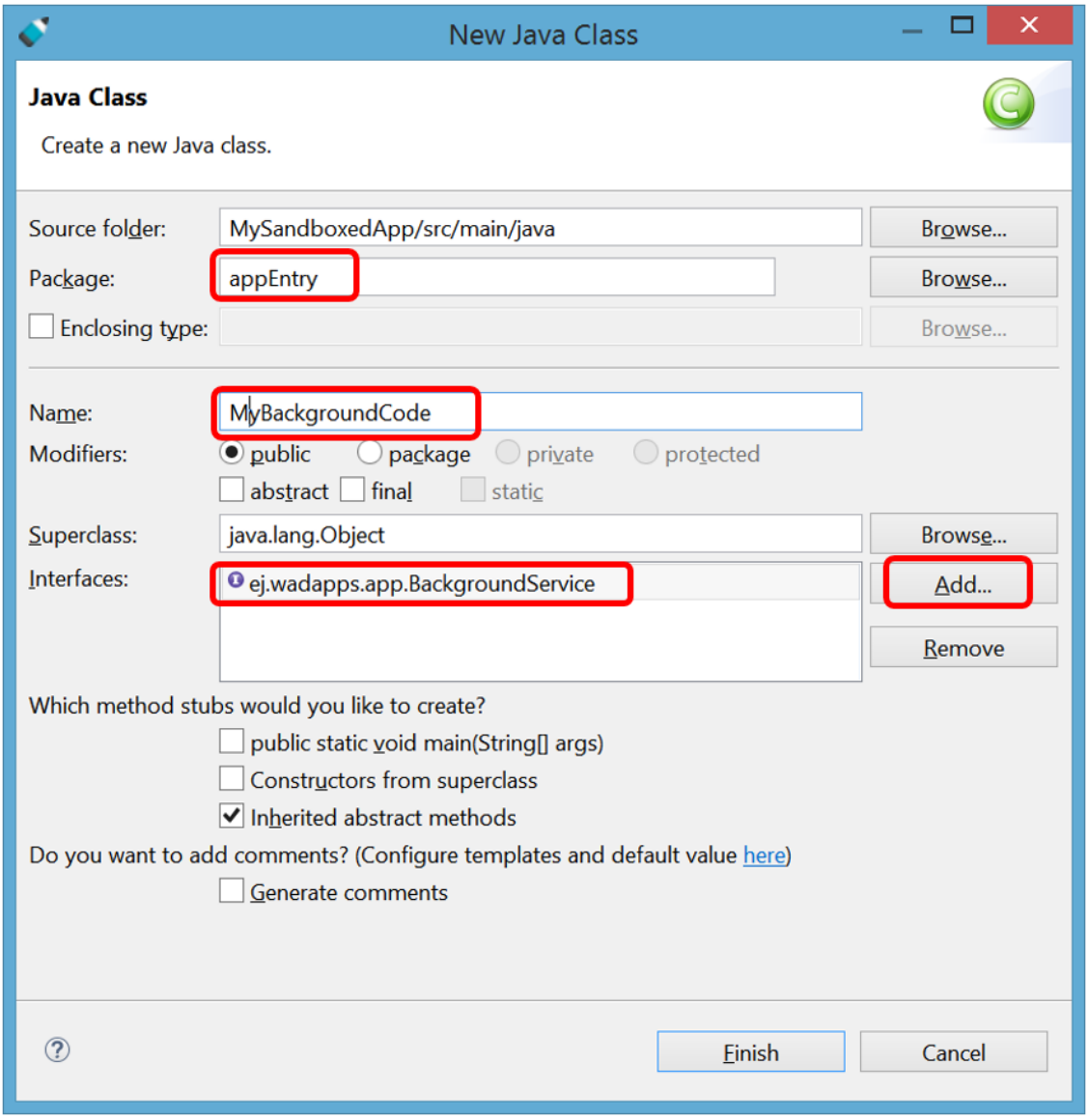
The classic *Hello World* application, which does not use the Graphical User Interface, is a good example of a BackgroundService entry point.

5.2.1.1. Classes

Create a new class in the `src/main/java` folder of the empty project:



Fill the new class with package information and give it a name that tells about its role as a Background Service. Notice that we have added the `ej.wadapps.app.BackgroundService` interface from the wadapps framework and that the class does not have a `main()` method.



The code to output messages on the console can now be added to the `onStart()` method, we also add a message to the `onStop()` method in order to follow the application's life cycle.

```
MyBackgroundCode.java   MANIFEST.MF

package appEntry;

import ej.wadapps.app.BackgroundService;

public class MyBackgroundCode implements BackgroundService {

    @Override
    public void onStart() {
        // TODO Auto-generated method stub
        System.out.println("MyBackgroundCode: Hello World");
    }

    @Override
    public void onStop() {
        // TODO Auto-generated method stub
        System.out.println("MyBackgroundCode: STOPPED");
    }
```

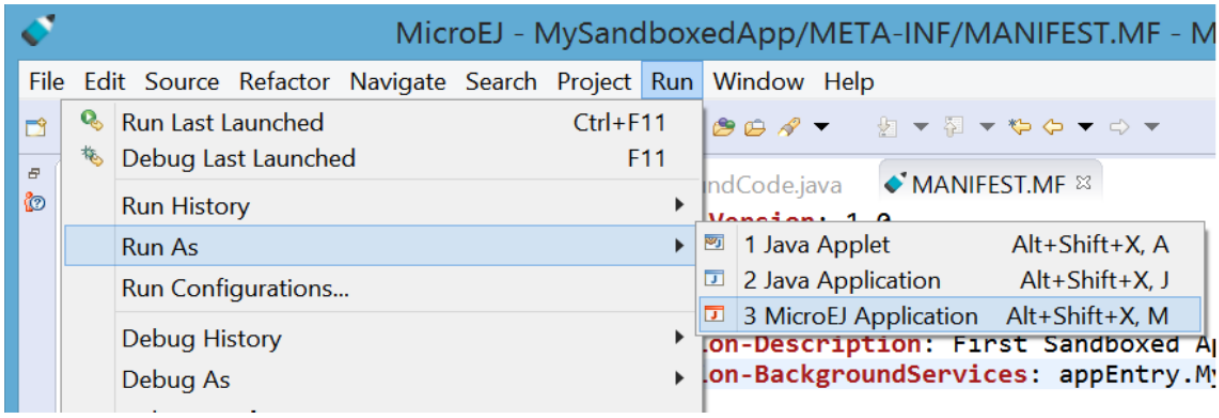
5.2.2. Manifest File Configuration

Our simple background application has one BackgroundService entrypoint. The `appEntry.MyBackgroundCode` class fully qualified name must be registered in the `Application-BackgroundServices` entry in the `MANIFEST.MF` file.

```
MicroEJ - MySandboxedApp/META-INF/MANIFEST.MF - MicroEJ® Studio
File Edit Source Refactor Navigate Search Project Run Window Help
Package Ex...   Type Hierar...
MySandboxedApp
  src/main/java
    appEntry
      MyBackgroundCode.java
  src/main/resources
  Ivy module.ivy [*]
  META-INF
    certificate
    libraries
    properties
    services
    MANIFEST.MF
  src
  CHANGELOG.md
  LICENSE.txt
  module.ivy
  README.md

MyBackgroundCode.java   MANIFEST.MF
Manifest-Version: 1.0
Application-Id: MyId
Application-Version: 0.1.0
Application-PrintableName: MyPrintable
Application-Description: First Sandboxed App
Application-BackgroundServices: appEntry.MyBackgroundCode

Console   Problems   Progress
'MySandboxedApp' EasyAnt Console
BUILD SUCCESSFUL

Total time: 0 seconds
---- Memory Details ----
  Used Memory  = 23MB

MANIFEST.MF - MySandboxedApp/META-INF
```

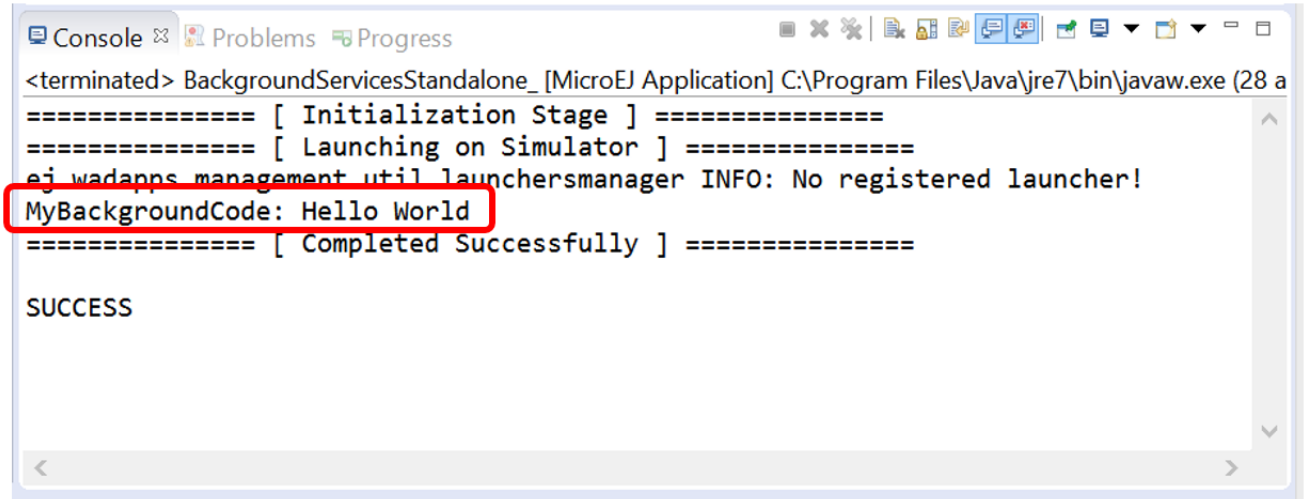
5.3. Test on a Virtual Device

To launch the application on the Simulator, select the `MySandboxedApp` project and in the MicroEJ top menu select `Run > Run As > MicroEJ Application`.

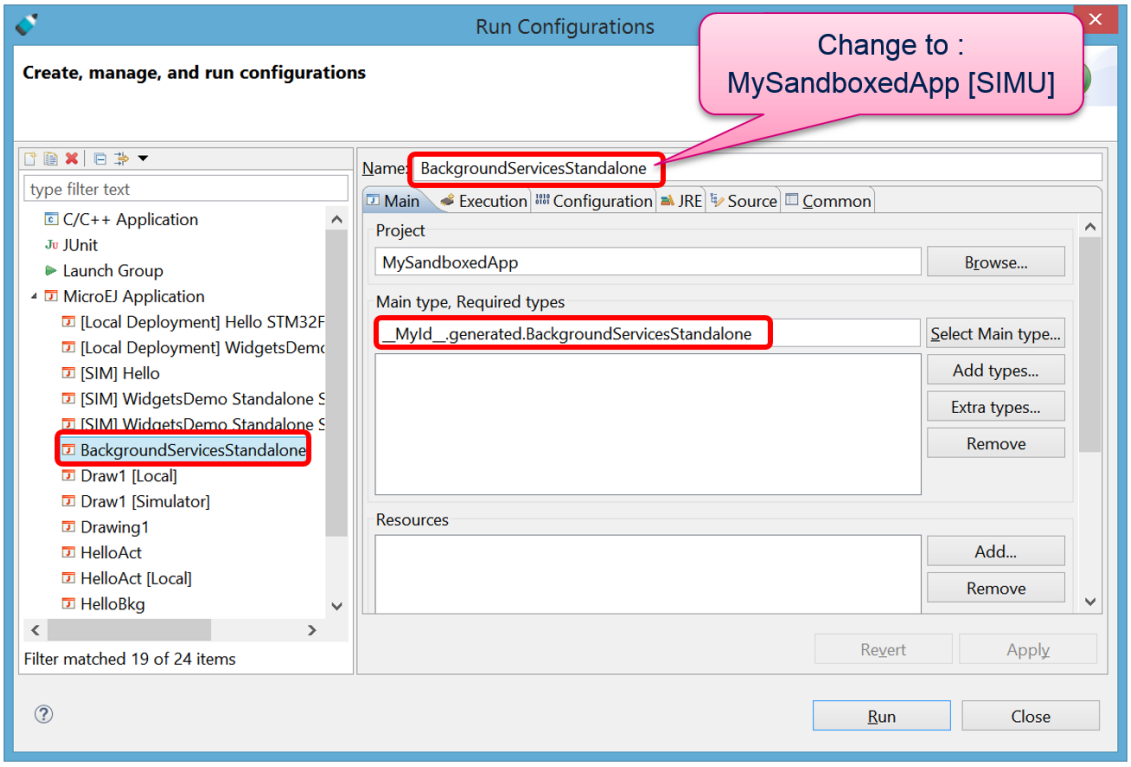


As this is the first launch for the application, the target must be set up for the launcher. If there is only one virtual device available in the MicroEJ repository, this virtual device is automatically selected. Otherwise, a popup window invites to select the one on which the application must be launched.

The application executes on the Simulator, as no graphic code is present the Simulator will not display its user interface and directly send output to the MicroEJ Studio Console window.

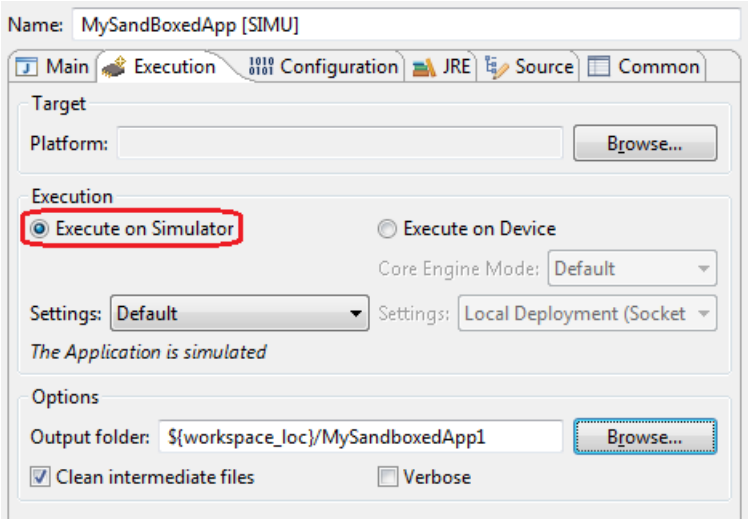


To edit the MicroEJ Launch Configuration automatically created by this first launch, open `Run > Run Configurations...` window. On the left panel open the `MicroEJ Application` category and select the `BackgroundserviceStandalone` run configuration.



The name of the run configuration was generated automatically from the name of the startup Class, you may change it to a more descriptive string (i.e. *MySandBoxedApp [SIMU]*). Note that the type selected for launching on simulator is the autogenerated main type for standalone application (see Section 3.4, “Standalone vs Sandboxed Application”).

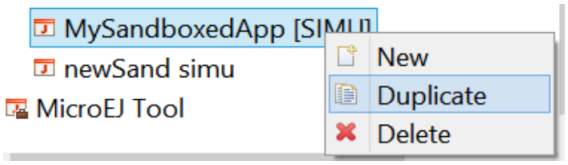
In the `Execution` tab of the run configuration, the Platform is set to the selected Virtual Device and execution mode set to `Execute on Simulator`.



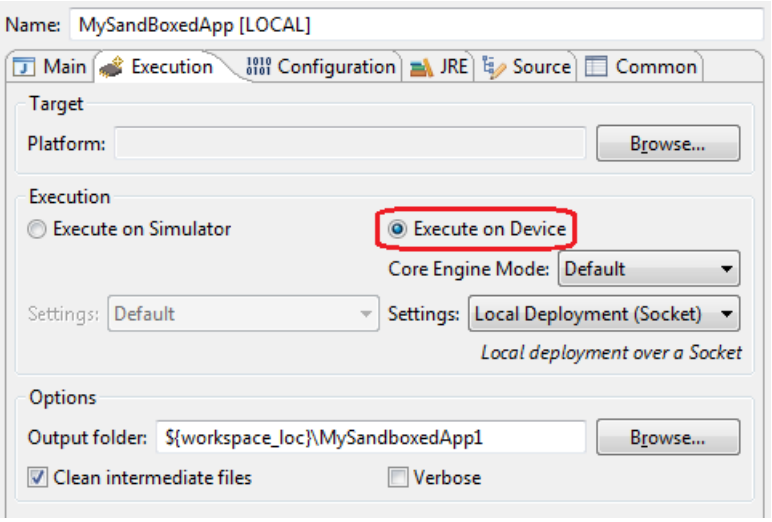
5.4. Test on Target Hardware

5.4.1. Create a Run Configuration for the Target Hardware

The run configuration for the target hardware is duplicated from the existing `MySandboxedApp [SIMU]` for the Simulator. In the left panel of `Run Configurations` window, right click on `MySandboxedApp [SIMU]` item and select `Duplicate`.

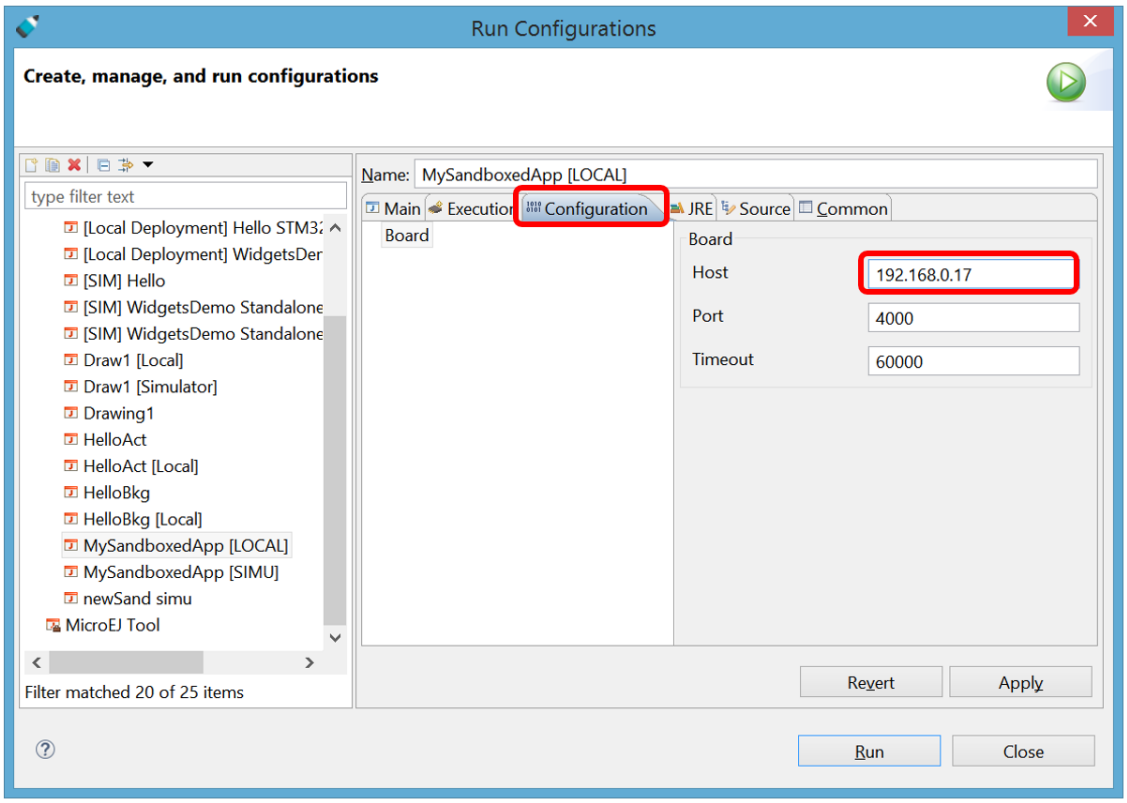


Rename the duplicated launcher to `MySandboxedApp [LOCAL]`, modify the execution mode to `Execute on Device` and check that `Settings` is set to `Local Deployment (Socket)`. Note that depending on the device capability, the virtual device may implement a local deployment over a Comm Port.



Next step is to configure the target hardware connection. Click on the `Configuration` tab and set the IP addess of the board. (See Section 9.1.1.1, “Board Connection Options” for Comm Port options alternative).

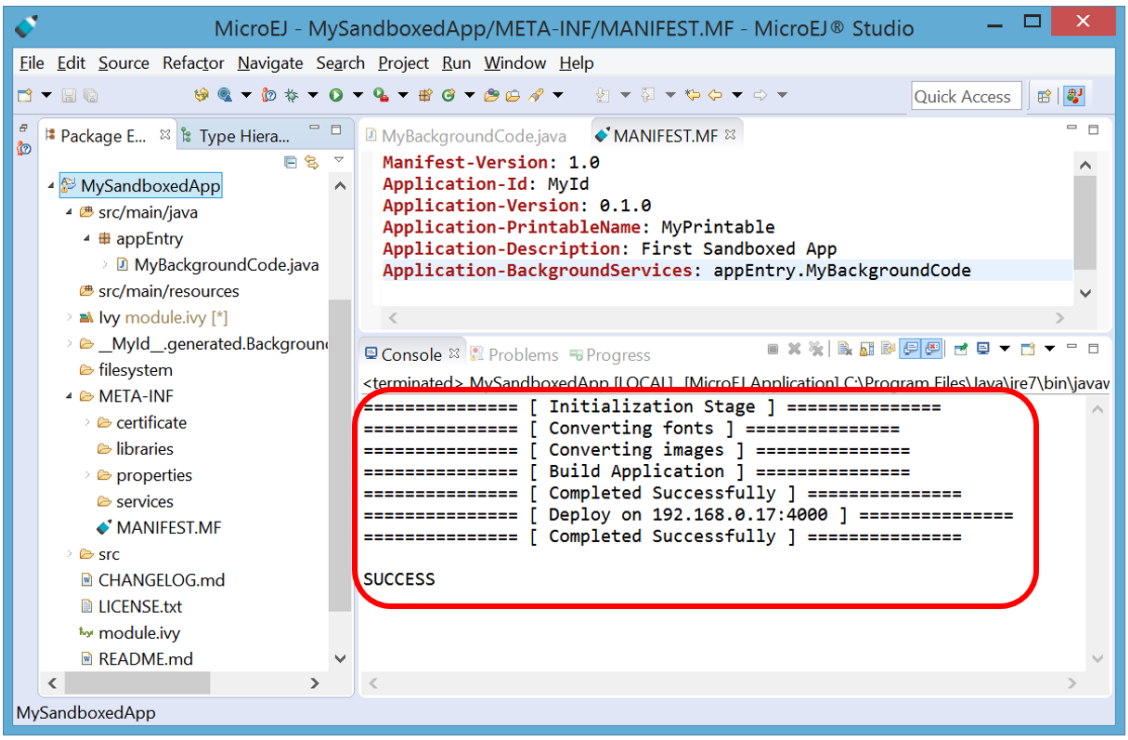
Enter the IP address on the `Host` field in the `Configuration` tab of the `MySandboxedApp [LOCAL]` launcher.



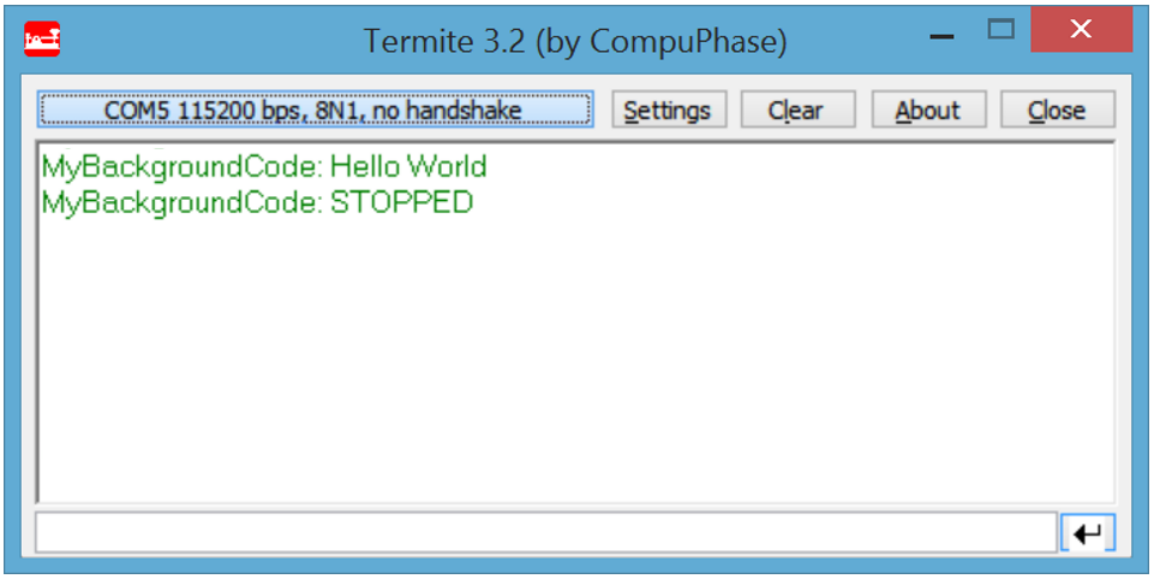
The run configuration is now ready for local deployment on the target.

5.4.2. Local Deployment on the Target Hardware

Run the `MySandboxedApp [LOCAL]` launcher, deployment steps are shown on the MicroEJ console.



The application is deployed on target hardware and automatic started. The debug traces show the life cycle of the sandboxed application.



Chapter 6. Activity Application

6.1. Develop an Activity Application

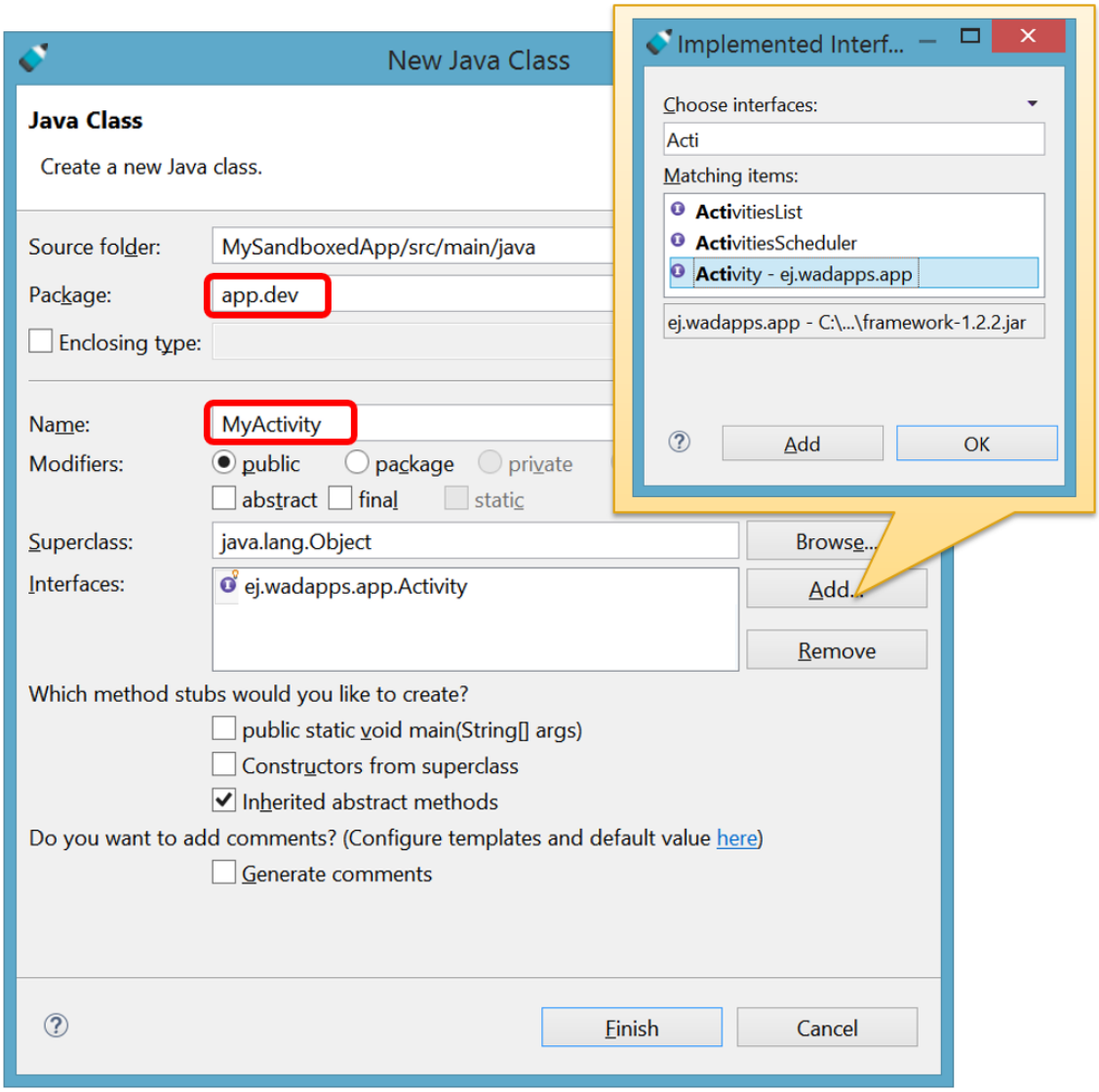
6.1.1. Create a Sandboxed Application Project

The first step to explore a sandboxed application structure is to create a new projet for the development of a graphical application.

See Section 5.1, “Create a Sandboxed Application Project” for creating a ready to use template project.

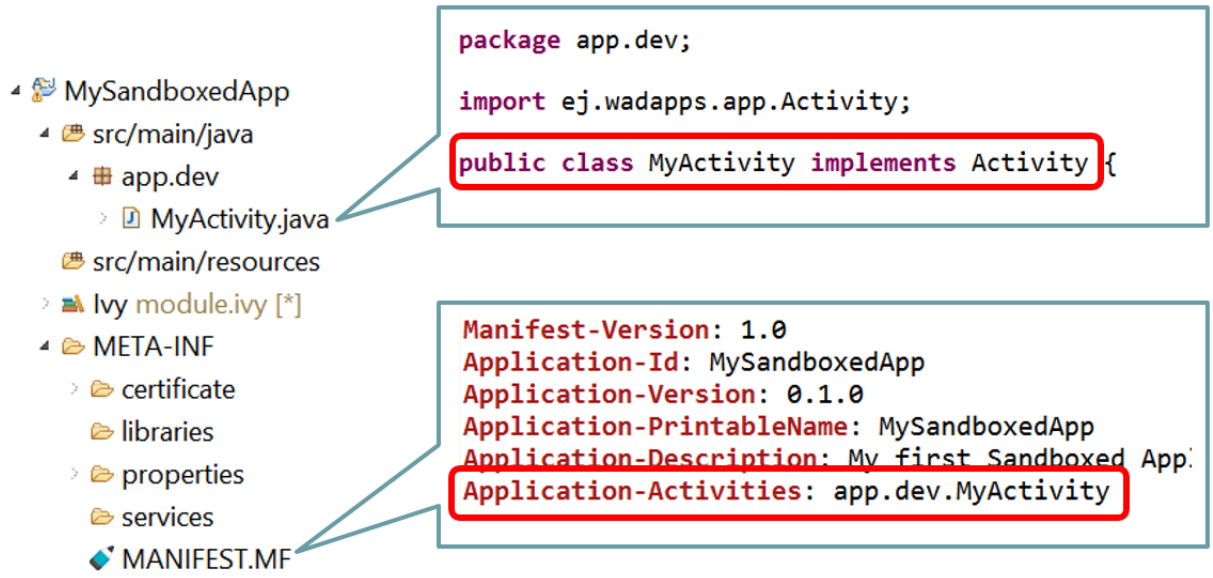
6.1.2. Create an Activity Implementation

A graphical application will have an Activity entry point to allow for screen sharing with other graphical applications. The first step is to create a class that will be the entry point of our sandboxed application. This class is located in the `src/main/java` folder and shall implement the `ej.wadapps.app.Activity` interface.



6.1.3. Update the Manifest File

Methods of the Activity interface handle the whole life cycle of a graphical application. The `app.dev.MyActivity` class fully qualified name must be registered in the `Application-Activities` entry in the `MANIFEST.MF` file.

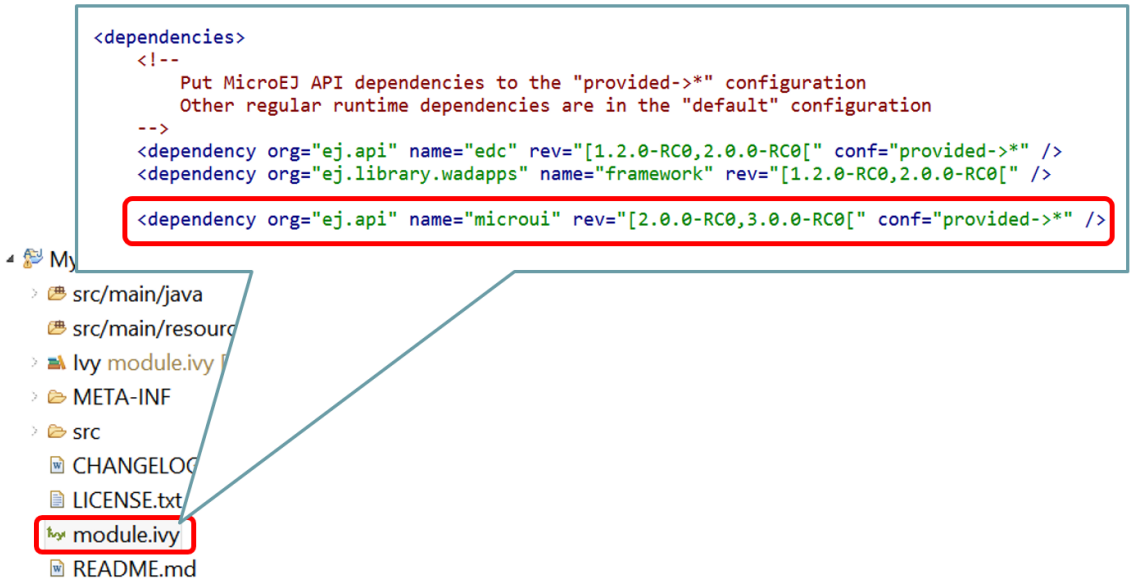


The `onStart()` method will do the job of initializing graphical objects.

6.1.4. Add Graphical Library Dependency

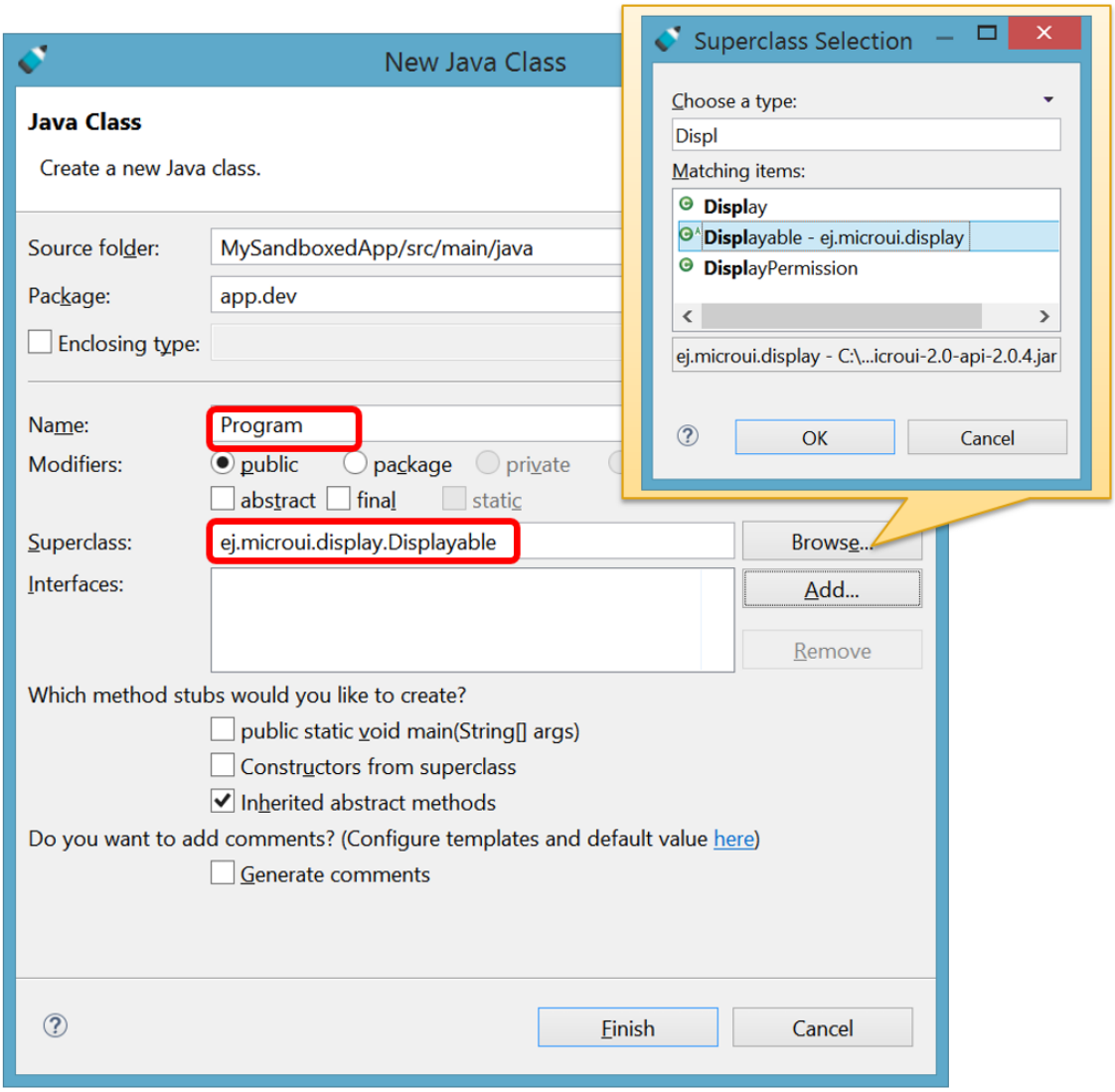
Since the application uses graphical objects, we have to complete `module.ivy` file to add a dependency to the corresponding GUI library: MicroUI (basic drawing elements).

The line describing this library is inserted in the `dependency` section of the `module.ivy` file. See Section 8.5, “Library Dependency Manager” for more information about classpath dependencies management.

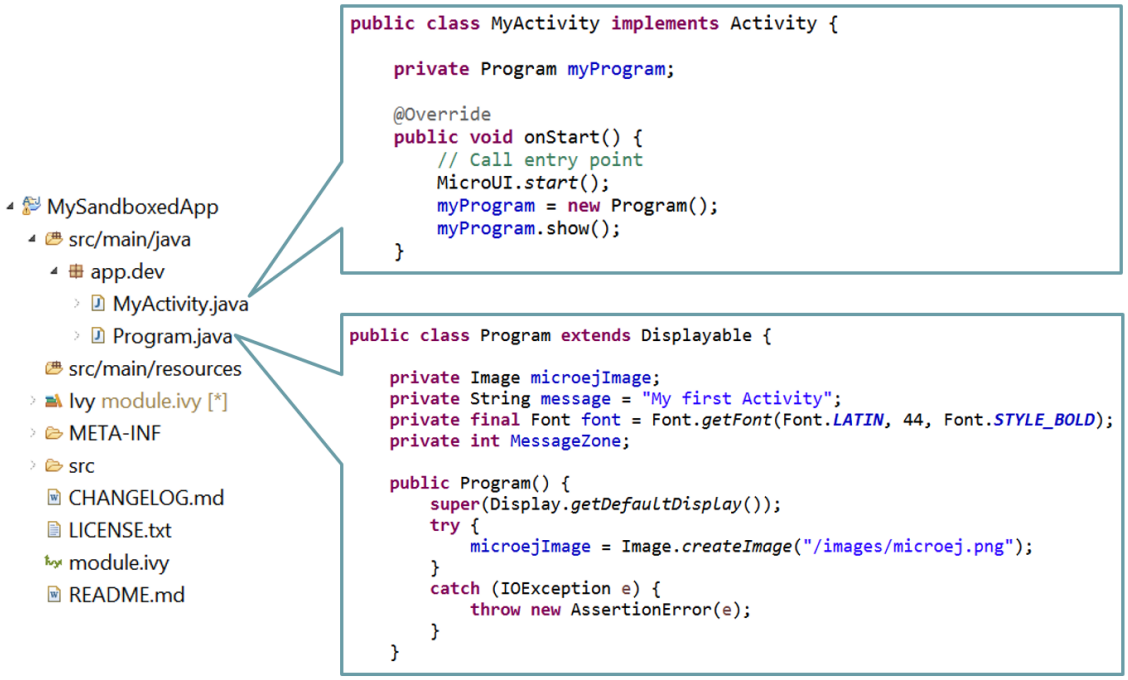


6.1.5. Implement a Graphical Class

A new class is added to the project for implementation of the graphical behavior, this class is named `app.dev.Program` and extends the `ej.microui.display.Displayable` MicroUI class.



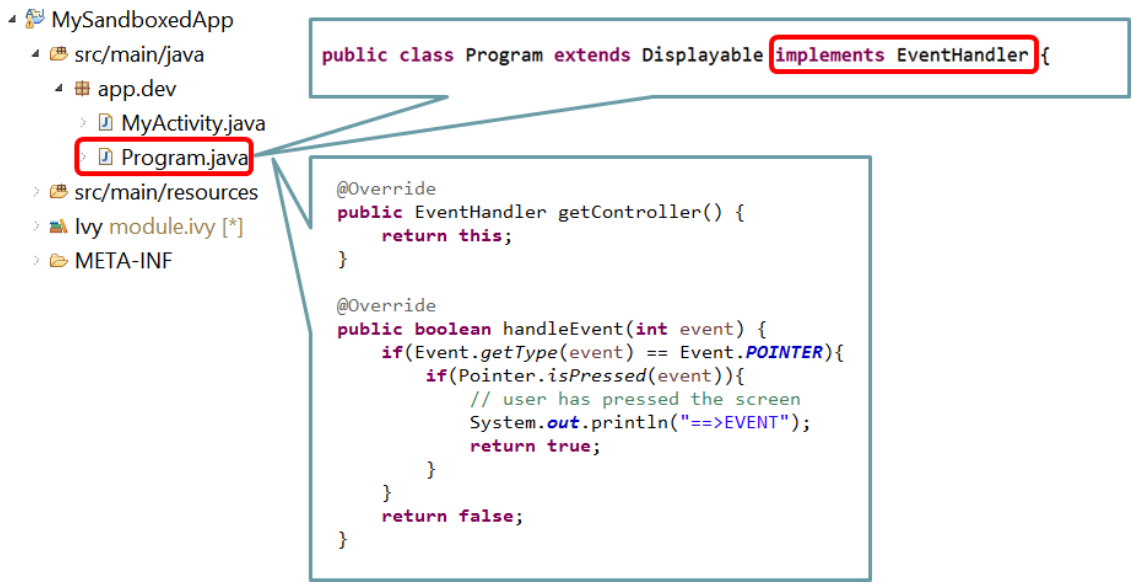
An instance of `app.dev.Program` is created in the Activity `onStart()` method, for this a dedicated constructor is added to the Program Class, with a reference to an image resource.



The paint method of the `ej.microui.display.Displayable` object is responsible for graphical output, the code of this method will first clear the screen by drawing a white rectangle, then compute layout infomation before displaying an image and a text.

```
public void paint(GraphicsContext g) {
    // clear screen
    g.setColor(Colors.WHITE);
    int width = getDisplay().getWidth();
    int height = getDisplay().getHeight();
    g.fillRect(0, 0, width, height);

    // compute margin
    int microejImageHeight = microejImage.getHeight();
    int fontHeight = font.getHeight();
    int margin = (height - microejImageHeight - fontHeight)/3;

    // draw MicroEJ image
    int y = microejImageHeight/2 + margin;
    g.drawImage(microejImage, width/2, y, GraphicsContext.HCENTER | GraphicsContext.VCENTER);

    // draw message
    int messageZone = microejImageHeight + 2*margin;
    y = messageZone + fontHeight/2;
    g.setColor(Colors.NAVY);
    g.setFont(font);
    g.drawString(message, width/2, y, GraphicsContext.HCENTER | GraphicsContext.VCENTER);
}
```

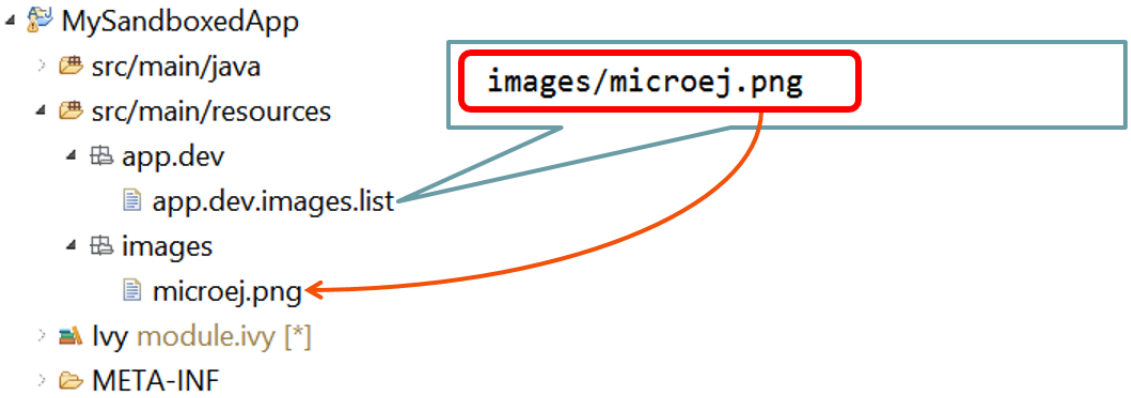
In order to react to user events, an EventHandler implementation is added to the `app.dev.Program` class. The implementation of `handleEvent()` method will test the pointer events in order to detect user actions.



6.2. Add Application Resources

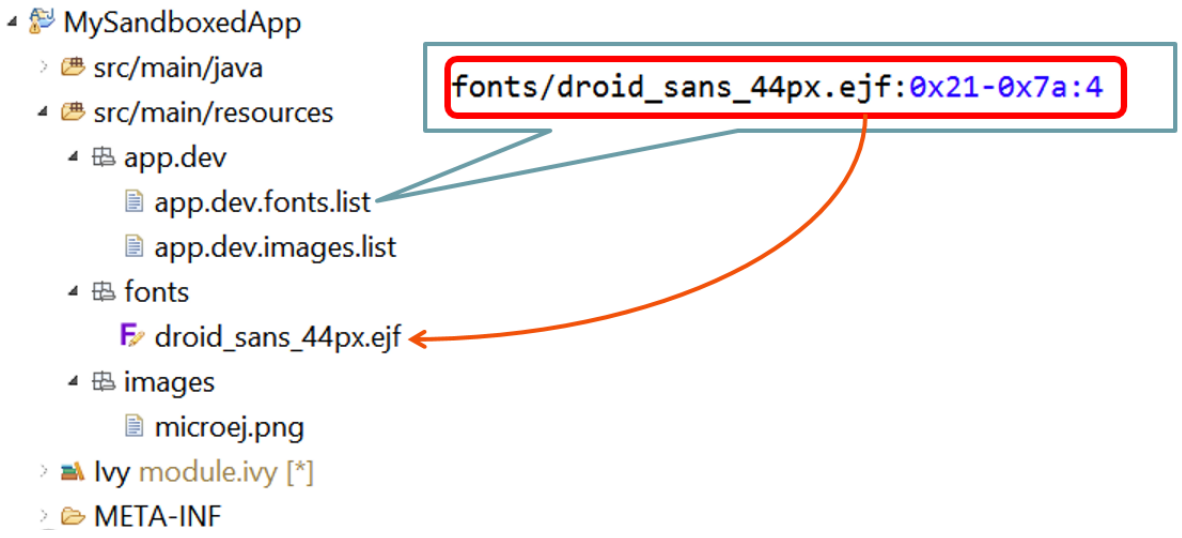
6.2.1. Add Images Resources

As shown in the previous section, the `app.dev.Program` class uses an image from a PNG file from image "microej.png" file which can be duplicated from the Hello sample. This file is embedded in the application by adding the `microej.png` file to a `src/main/resources/images` folder, and by adding a reference to this file in the `app.gui.images.list` file added to the `src/main/resources/app/dev` folder (see Section 8.3.6, “Images” for images list files specification).



6.2.2. Add Fonts Resources

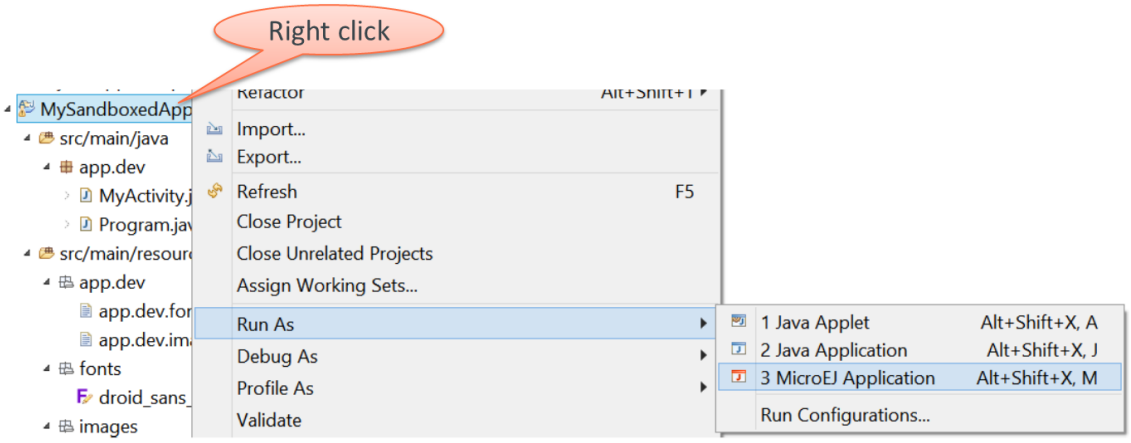
A dedicated large font will be used to display the text on the Button widget, the font will be embedded in the application by using the same technique as the image file. The `droid_sans_44px.ejf` font file is copied from Hello sample to a "`src/main/resources/fonts` folder, and a new `app.gui.fonts.list` file containing the font reference is created in the `src/main/resources/app/dev` folder" (see Section 8.3.7, “Fonts” for fonts list files specification).



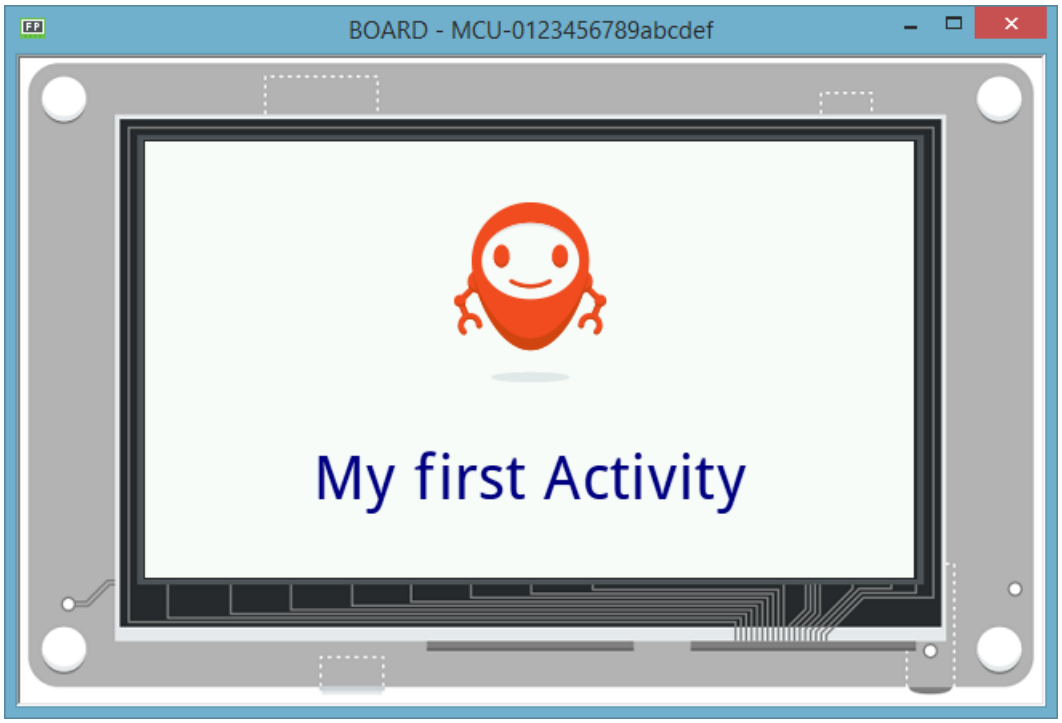
6.3. Test the Application on Simulator

6.3.1. Create a Run Configuration

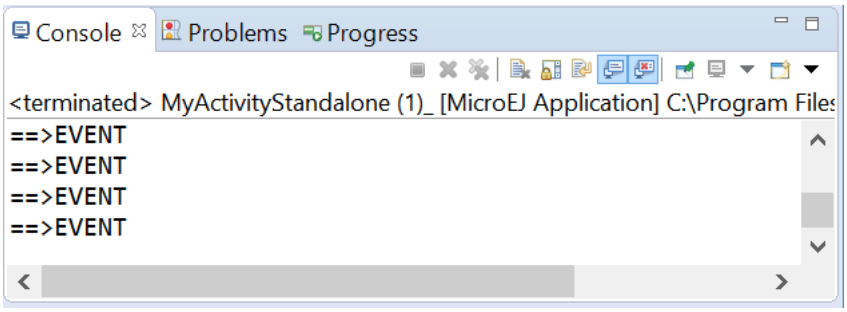
To rapidly test the application, right click on project's name and select `Run as > MicroEJ Application`.



The simulator will launch with the following graphical result:



Clicking on the screen will produce the following result in the MicroEJ Studio Console:



Chapter 7. Shared Interfaces

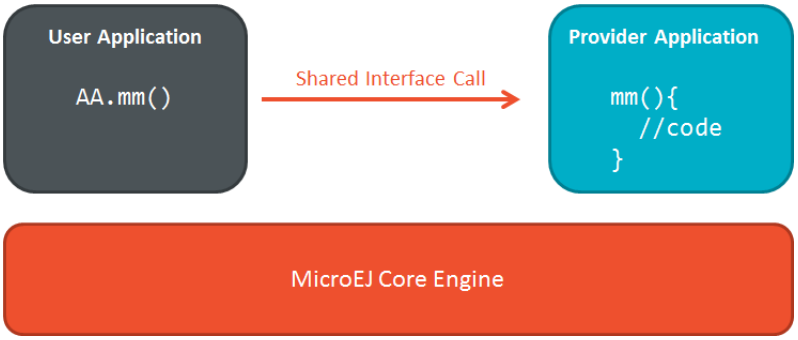
7.1. Principle

The Shared Interface mechanism provided by MicroEJ core engine is an object communication bus based on plain Java interfaces where method calls are allowed to cross MicroEJ Sandboxed applications boundaries. The Shared Interface mechanism is the cornerstone for designing reliable Service Oriented Architectures on top of MicroEJ. Communication is based on the sharing of interfaces defining APIs (Contract Oriented Programming).

The basic schema:

- A provider application publishes an implementation for a shared interface into a system registry (see Section 7.4, “System Registries”).
- A user application retrieves the implementation from the system registry and directly calls the methods defined by the shared interface

Figure 7.1. Shared Interface Call Mechanism



7.2. Shared Interface Creation

Creation of a shared interface follows three steps:

- Interface definition
- Proxy implementation
- Interface registration

7.2.1. Interface Definition

The definition of a shared interface starts by defining a standard Java interface.

```
package mypackage;
public interface MyInterface{
 void foo();
}
```

To declare an interface as a shared interface, it must be registered in a shared interfaces identification file. A shared interface identification file is an XML file with the `.si` suffix with the following format:

```
<sharedInterfaces>
 <sharedInterface name="mypackage.MyInterface"/>
</sharedInterfaces>
```

Shared interface identification files must be placed at the root of a path of the application classpath. For a MicroEJ Sandboxed application project, it is typically placed in `src/main/resources` folder.

Some restrictions apply to shared interface compared to standard java interfaces:

- Types for parameters and return values must be transferable types
- Thrown exceptions must be classes owned by the MicroEJ Firmware

7.2.2. Transferable Types

In the process of a cross-application method call, parameters and return value of methods declared in a shared interface must be transferred back and forth between application boundaries.

Figure 7.2. Shared Interface Parameters Transfer

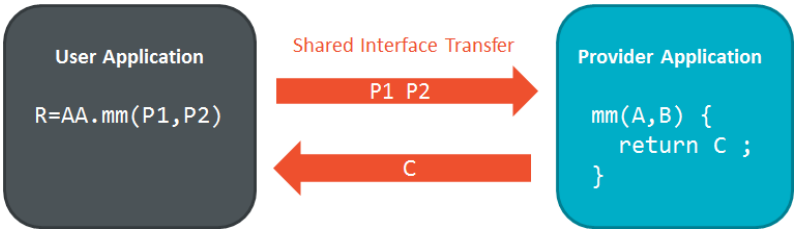


Table 7.1, “Shared Interface Types Transfer Rules” describes the rules applied depending on the element to be transferred.

Table 7.1. Shared Interface Types Transfer Rules

Type	Owner	Instance Owner	Rule
Base type	N/A	N/A	Passing by value. (`boolean`, `byte`, `short`, `char`, `int`, `long`, `double`, `float`)

Type	Owner	Instance Owner	Rule
Any Class, Array or Interface	MicroEJ Firmware	MicroEJ Firmware	Passing by reference
Any Class, Array or Interface	MicroEJ Firmware	Application	MicroEJ Firmware specific or forbidden
Array of base types	Any	Application	Clone by copy
Arrays of references	Any	Application	Clone and transfer rules applied again on each element
Shared Interface	Application	Application	Passing by indirect reference (Proxy creation)
Any Class, Array or Interface	Application	Application	Forbidden

Objects created by an application which class is owned by MicroEJ Firmware can be transferred to an other application if this has been authorized by the firwmare. The list of eligible types that can be transferred is firwmare specific, so you have to consult the firmware specification. Table 7.2, “MicroEJ Evaluation Firmware Example of Transfer Types” lists firmware types allowed to be transferred through a shared interface call. When an argument transfer is forbidden, the call is abruptly stopped and a `java.lang.IllegalAccessError` is thrown by MicroEJ Core Engine.

Table 7.2. MicroEJ Evaluation Firmware Example of Transfer Types

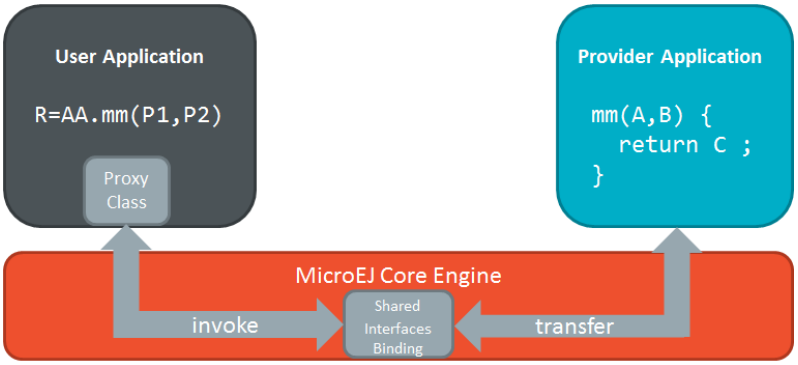
Type	Rule
`java.lang.String`	Clone by copy
`java.io.InputStream`	Proxy reference creation
`java.util.Map<String,String>`	Clone by deep copy

7.2.3. Proxy Class Implementation

The Shared Interface mechanism is based on automatic proxy objects created by the underlying MicroEJ Core Engine, so that each application can still be dynamically stopped and uninstalled. This offers a reliable way for users and providers to handle the relationship in case of a broken link.

Once a shared interface has been declared as shared interface, a dedicated implementation is required (called the Proxy class implementation). Its main goal is to perform the remote invocation and provide a reliable implementation regarding the interface contract even if the remote application fails to fulfill its contract (unexpected exceptions, application killed...). The MicroEJ Core Engine will allocate instances of this class when an implementation owned by an other application is being transferred to this application.

Figure 7.3. Shared Interfaces Proxy Overview



A proxy class is implemented and executed on the client side, each method of the implemented interface must be defined according to the following pattern:

```
package mypackage;

public class MyInterfaceProxy extends Proxy<MyInterface> implements
 MyInterface {

 @Override
 public void foo(){
  try {
   invoke(); // perform remote invocation
  } catch (Throwable e) {
   e.printStackTrace();
  }
 }
}
```

Each implemented method of the proxy class is responsible for performing the remote call and catching all errors from the server side and to provide an appropriate answer to the client application call according to the interface method specification (contract). Remote invocation methods are defined in the super class `ej.kf.Proxy` and are named `invokeXXX()` where XXX is the kind of return type. As this class is part of the application, the application developer has the full control on the Proxy implementation and is free to insert additional code such as logging calls and errors for example.

Table 7.3. Proxy Remote Invocation Built-in Methods

Invocation Method	Usage
void invoke()	Remote invocation for a proxy method that returns void
Object invokeRef()	Remote invocation for a proxy method that returns a reference
boolean invokeBoolean(), byte invokeByte(), char invokeChar(), short in-	Remote invocation for a proxy method that returns a base type

Invocation Method	Usage
vokeShort(), int invokeInt(), long invokeLong(), double invokeDouble(), float invokeFloat()	

7.3. Shared Interface Example

The sample code hereafter shows an example of a Shared Interface named MyOutput with two methods println and nbExec.

```
public interface MyOutput {

    /**
     * Print function.
     *
     * @param str
     *            The string to print.
     * @throws IOException
     *             Throw an IOException when the service is not available.
     */
    void println(String str) throws IOException;

    /**
     * Get the number of time println has been executed.
     *
     * @return The number of time println has been executed.
     */
    int nbExec();
}
```

With this interface we will transform a simple "Hello" project into a print client using a shared interface provided by a server application.

7.3.1. Write the Proxy Implementation

An example of a Proxy class for the MyOutput Shared Interface is shown hereafter:

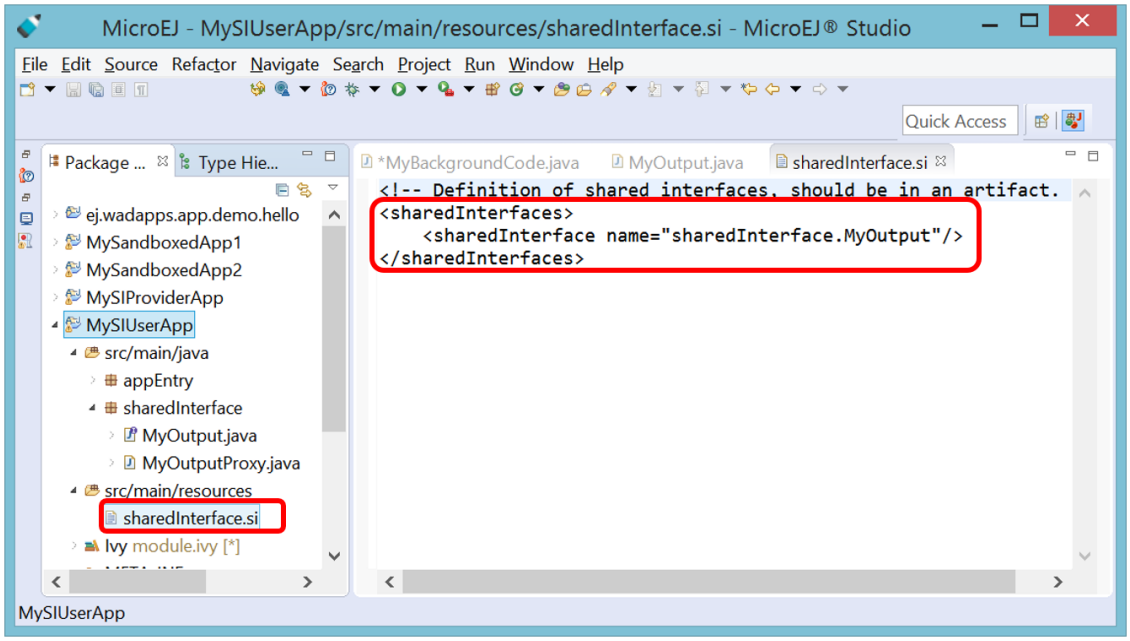
```
public class MyOutputProxy extends Proxy<MyOutput> implements MyOutput {

    @Override
    /**
     * Proxy to the interface implementation of println.
     */
    public void println(String str) throws IOException{
        try{
            invoke();                   // Invoke the implementation
        }
        catch(Throwable e){             // For any exception during the execution of the
            throw new IOException();    // function (server side), the catch it here.
        }
    }

    @Override
    /**
     * Proxy to the interface implementation of nbExec.
     */
    public int nbExec(){
        try{
            return invokeInt();
        }
        catch(Throwable e){
            return -1;                  // return a default value
        }
    }
}
```

7.3.2. Prepare the Shared Interface Projects

To migrate the simple "Hello" application `MySandboxedApp` into a Shared Interface sample, first duplicate the `MySandboxedApp` project to a `MySIUserApp` project and add three files to this project:

- `MyOutput.java` to `src/main/java` folder in the `sharedInterface` package
- `MyOutputProxy.java` to `src/main/java` folder in the `sharedInterface` package
- `sharedInterface.si` to `src/main/resources` folder

The resulting modifications should appear as follows in MicroEJ Studio. Note the XML syntax of the `.si` declaration file containing the full qualified name of the Shared Interface type.



Once the `MySIUserApp` project is updated, duplicate it to a `MySIProviderApp` project, both projects have the same content at this point.

Make sure to update the `Application-Id` fields of the `MANIFEST.MF` files of both projects so that they are different from each other to prevent one application from overwriting the other when deploying on target.

Figure 7.4. **`MANIFEST.MF`** file content for Shared Interface Provider application project

```
Manifest-Version: 1.0
Application-Id: MySIProvider
Application-Version: 0.1.0
Application-PrintableName: MySIProvider
Application-Description: MySIProvider App
Application-BackgroundServices: appEntry.MyBackgroundCode
```

Figure 7.5. **`MANIFEST.MF`** file content for Shared Interface User application project

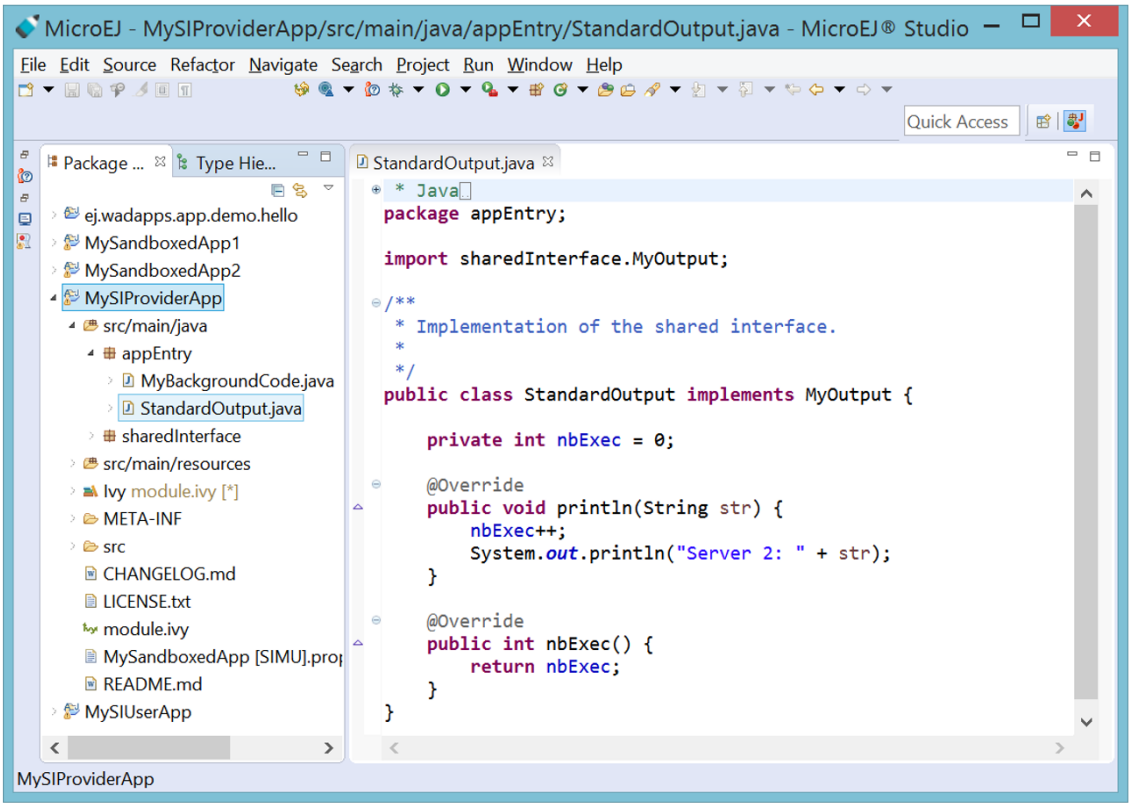
```
Manifest-Version: 1.0
Application-Id: MySIUser
Application-Version: 0.1.0
Application-PrintableName: MySIUser
Application-Description: MySIUser App
Application-BackgroundServices: appEntry.MyBackgroundCode
```

We will now specialize the Background Services.

7.3.3. Implement the Provider Side

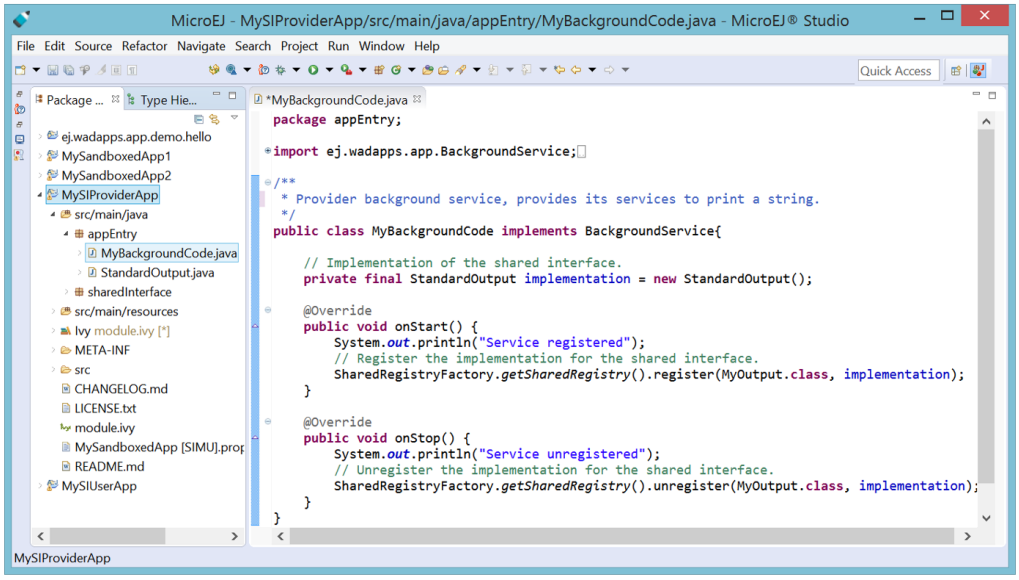
7.3.3.1. Create the Provider Implementation Class

The provider side implementation of a Shared Interface follows the standard rule of java language for interface implementation. Add a new class with the name `StandardOutput` to the `src/main/java` folder, this class implements the `sharedInterface.MyOutput` interface.



7.3.3.2. Register the Provider as a Shared Interface Implementation

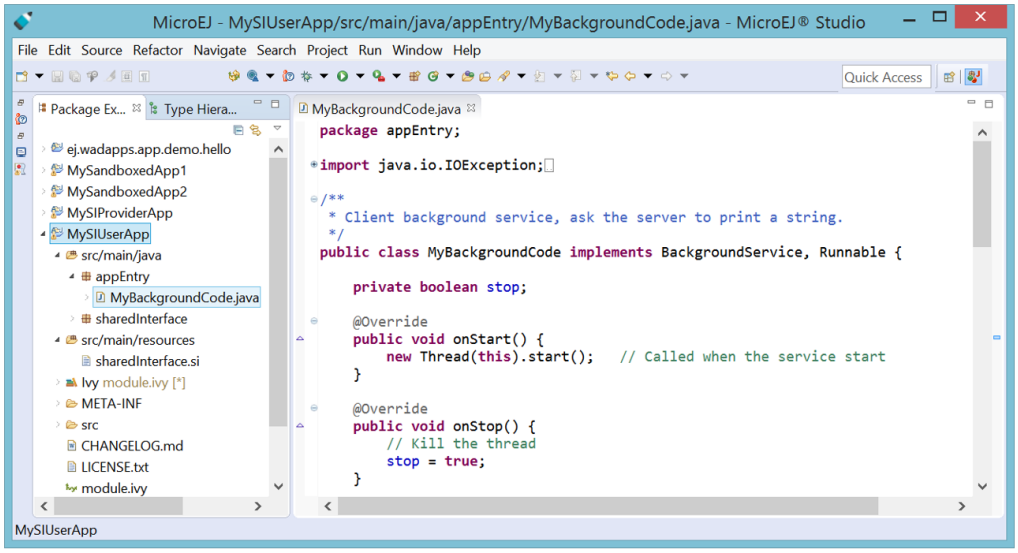
In order to expose or hide its implementation of the Shared Interface, the provider uses the MicroEJ Registry service with the help of the `ej.wadapps.registry.SharedRegistryFactory` object.



7.3.4. Implement the User Side

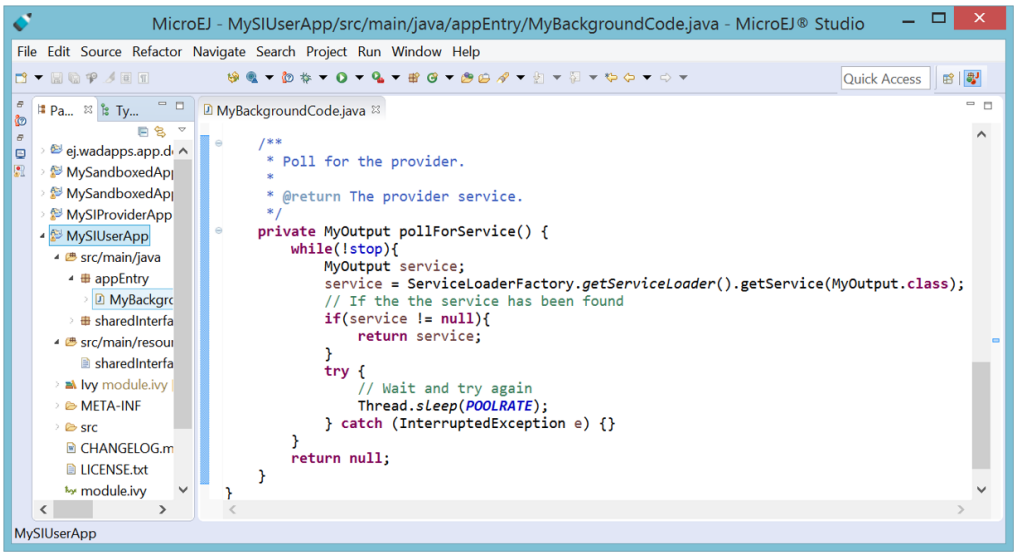
7.3.4.1. Write the User Behaviour

In order to generate periodic activity on the shared interface, the user application declares a Background Service that runs a cyclic thread.



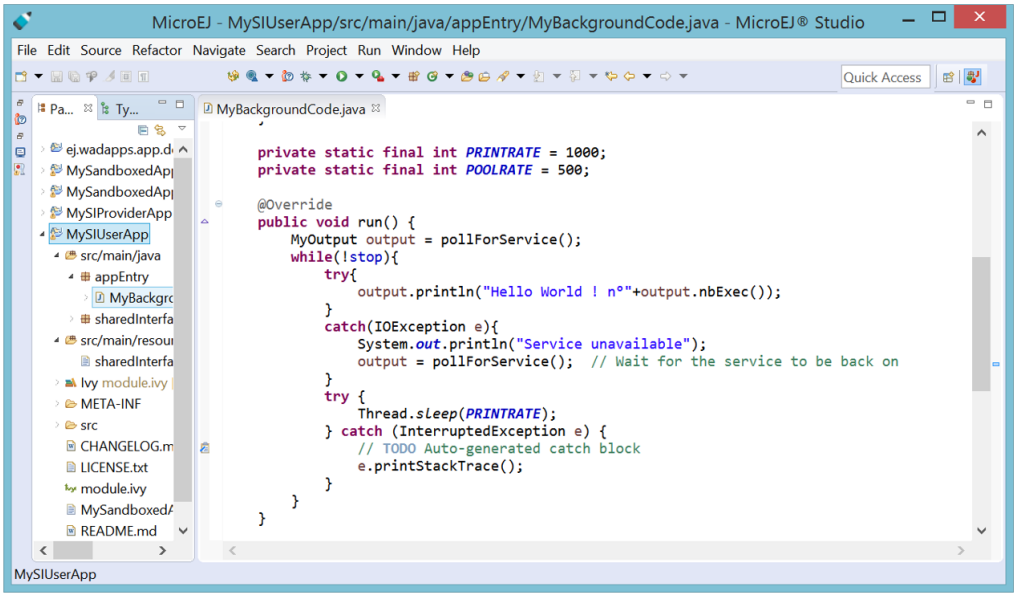
7.3.4.2. Get the Provider Service Reference

After retrieving the `ServiceLoader` instance, the user gets a local reference to the registered provider. (see Section 3.3, “Services Usage” for more informations on services references).



7.3.4.3. Call the Provider Service

With a valid reference to the provider service, the user calls the `MyOutput` interface methods.



As the session is loosely coupled, the call is performed with an exception handler to prevent from a change in the provider status. If the call fails, the user starts polling the service loader again to retrieve a new valid instance.

To show the communication between `MySIUserApp` and `MySIProviderApp`, the two applications must be locally deployed on a MicroEJ-ready device (see Chapter 2, *MicroEJ Studio Getting Started*). The messages will be displayed on the standard output.

7.4. System Registries

MicroEJ provides system registries that allow applications to publish/retrieve shared interfaces implementations. When a shared interface instance is published into such kind of registry, the registry makes it accessible to other applications. MicroEJ provides two system registries:

- The Wadapps framework shared registry (`ej.wadapps.registry.SharedRegistry`) is dedicated to sharing service related interfaces.

  Services can be retrieved using the following API `ej.components.dependencyinjection.ServiceLoader.getService(Class)`. See Section 3.3, “Services Usage” for how to retrieve services.

- The ECOM device manager (`ej.ecom.DeviceManager`) registry is dedicated to sharing peripheral extensions related interfaces.

  Applications can register device extensions that are dynamically discovered using the following API `ej.ecom.DeviceManager.register(Class<Device>, Device)`. See `ECOM` foundation library API javadoc for more information.

Chapter 8. MicroEJ Classpath

MicroEJ applications run on a target device and their footprint is optimized to fulfill embedded constraints. The final execution context is an embedded device that may not even have a file system. Files required by the application at runtime are not directly copied to the target device, they are compiled to produce the application binary code which will be executed by MicroEJ core engine.

As a part of the compile-time trimming process, all types not required by the embedded application are eliminated from the final binary.

MicroEJ Classpath is a developer defined list of all places containing files to be embedded in the final application binary. MicroEJ Classpath is made up of an ordered list of paths. A path is either a folder or a zip file, called a JAR file (JAR stands for Java ARchive).

- Section 8.1, “Application Classpath” explains how the MicroEJ classpath is built from a MicroEJ application project.
- Section 8.2, “Classpath Load Model” explains how the application content is loaded from MicroEJ Classpath.
- Section 8.3, “Classpath Elements” specifies the different elements that can be declared in MicroEJ Classpath to describe the application content.
- Section 8.4, “Foundation vs Add-On Libraries” explains the different kind of libraries that can be added to MicroEJ Classpath.
- Finally, Section 8.5, “Library Dependency Manager” shows how to manage libraries dependencies in MicroEJ.

8.1. Application Classpath

The following schema shows the classpath mapping from a MicroEJ application project to the MicroEJ Classpath ordered list of folders and JAR files. The classpath resolution order (left to right) follows the project appearance order (top to bottom).

Figure 8.1. MicroEJ Application Classpath Mapping

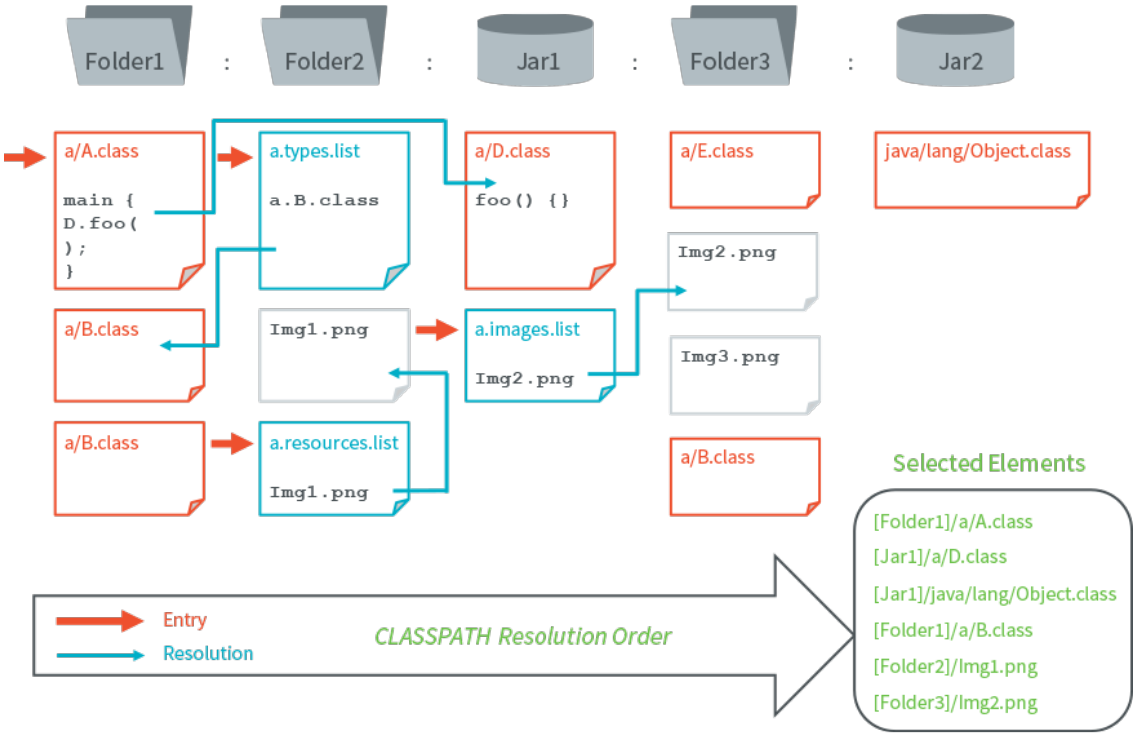
8.2. Classpath Load Model

A MicroEJ Application classpath is created via the loading of :

- an entry point type
- all `*.[extension].list` files declared in a MicroEJ Classpath.

The different elements that constitute an application are described in Section 8.3, “Classpath Elements”. They are searched within MicroEJ Classpath from left to right (the first file found is loaded). Types referenced by previously loaded MicroEJ Classpath elements are loaded transitively.

Figure 8.2. Classpath Load Principle



8.3. Classpath Elements

The MicroEJ Classpath contains the following elements:

- An entrypoint described in section Section 8.3.1, “Application Entry Points”
- Types in `.class` files, described in section Section 8.3.2, “Types”
- Raw resources, described in section Section 8.3.3, “Raw Resources”
- Immutables Object data files, described in Section Section 8.3.4, “Immutable Objects”
- Images and Fonts resources
- `*.[extension].list` files, declaring contents to load. Supported list file extensions and format is specific to declared application content and is described in the appropriate section.

8.3.1. Application Entry Points

MicroEJ application entry point is a class that contains a `public static void main(String[])` method. In case of MicroEJ Sandboxed Application, this entry point is automatically generated by MicroEJ Studio from declared Activity and/or BackgroundService types. In case of a MicroEJ Standalone application, this has to be defined by the user.

8.3.2. Types

MicroEJ types (classes, interfaces) are compiled from source code (`.java`) to classfiles (`.class`). When a type is loaded, all types dependencies found in the classfile are loaded (transitively).

A type can be declared as a *Required type* in order to enable the following usages:

- to be dynamically loaded from its name (with a call to `Class.forName(String)`)
- to retrieve its fully qualified name (with a call to `Class.getName()`)

A type that is not declared as a *Required type* may not have its fully qualified name (FQN) embedded. Its FQN can be retrieved using the stack trace reader tool (see Section 10.3, “Stack Trace Reader”).

Required Types are declared in MicroEJ Classpath using `*.types.list` files. The file format is a standard Java properties file, each line listing the fully qualified name of a type. Example:

Example 8.1. Required Types **`*.types.list`** File Example

```
# The following types are marked as MicroEJ Required Types
com.mycompany.MyImplementation
java.util.Vector
```

8.3.3. Raw Resources

Raw resources are binary files that need to be embedded by the application so that they may be dynamically retrieved with a call to `Class.getResourceAsStream(java.io.InputStream)`. Raw Resources are declared in MicroEJ Classpath using `*.resources.list` files. The file format is a standard Java properties file, each line is a relative `/` separated name of a file in MicroEJ Classpath to be embedded as a resource. Example:

Example 8.2. Raw Resources **`*.resources.list`** File Example

```
# The following resource is embedded as a raw resource
com/mycompany/MyResource.txt
```

8.3.4. Immutable Objects

Immutables objects are regular read-only objects that can be retrieved with a call to `ej.bon.Immutables.get(String)`. Immutables objects are declared in files called *immutable objects data files*, which format is described in the `[B-ON]` specification (`http://e-s-r.net`). Immutables objects data files are declared in MicroEJ Classpath using `*.immutables.list` files. The file format is a standard Java properties file, each line is a `/` separated name of a relative file in MicroEJ Classpath to be loaded as an Immutable objects data file. Example:

Example 8.3. Immutable Objects Data Files **`*.immutables.list`** File Example

```
# The following file is loaded as an Immutable objects data files
com/mycompany/MyImmutables.data
```

8.3.5. System Properties

System Properties are key/value string pairs that can be accessed with a call to `System.getProperty(String)`. System properties are declared in MicroEJ Classpath `*.properties.list` files. The file format is a standard Java properties file. Example:

Example 8.4. System Properties **`*.properties.list`** File Example

```
# The following property is embedded as a System property
com.mycompany.key=com.mycompany.value
```

8.3.6. Images

8.3.6.1. Overview

Images are graphical resources that can be accessed with a call to `ej.microui.display.Image.createImage()`. To be displayed, these images have to be converted from their source format to the display raw format. The conversion can either be done at :

- build-time (using the image generator tool)
- run-time (using the relevant decoder library)

Images that must be processed by the image generator tool are declared in MicroEJ Classpath `*.images.list` files. The file format is a standard Java properties file, each line representing a `/` separated resource path relative to the MicroEJ classpath root referring to a standard image file (e.g. `.png`, `.jpg`). The resource may be followed by an optional parameter (separated by a `:`) which defines and/or describe the image output file format (raw format). When no option is specified, the image is embedded as-is and will be decoded at run-time (although listing files without format specifier has no impact on the image generator processing, it is advised to specify them in the `*.images.list` files anyway, as it makes the run-time processing behavior explicit). Example:

Figure 8.3. Image Generator **`*.images.list`** File Example

```
# The following image is embedded
# as a PNG resource (decoded at run-time)
com/mycompany/MyImage1.png

# The following image is embedded
# as a 16 bits format without transparency (decoded at build-time)
com/mycompany/MyImage2.png:RGB565

# The following image is embedded
# as a 16 bits format with transparency (decoded at build-time)
com/mycompany/MyImage3.png:ARGB1555
```

8.3.6.2. Output Formats

8.3.6.2.1. No Compression

When no output format is set in the images list file, the image is embedded without any conversion / compression. This allows you to embed the resource as well, in order to keep the source image characteristics (compression, bpp etc.). This option produces the same result as specifiying an image as a resource in the MicroEJ launcher.

Advantages:

- Preserves the image characteristics.

Disadvantages:

- Requires an image runtime decoder.
- Requires some RAM in which to store the decoded image

Figure 8.4. Unchanged Image Example

```
image1
```

8.3.6.2.2. Display Output Format

This format encodes the image into the exact display memory representation. If the image to encode contains some transparent pixels, the output file will embed the transparency according to the display's implementation capacity. When all pixels are fully opaque, no extra information will be stored in the output file in order to free up some memory space.

Advantages:

- Drawing an image is very fast.
- Supports alpha encoding.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels.

Figure 8.5. Display Output Format Example

```
image1:display
```

8.3.6.2.3. Generic Output Formats

Depending on the target hardware, several generic output formats are available. Some formats may be directly managed by the BSP display driver. Refer to the platform specification to retrieve the list of natively supported formats.

Advantages:

- The pixels layout and bits format are standard, so it is easy to manipulate these images on the C-side.
- Drawing an image is very fast when the display driver recognizes the format (with or without transparency).
- Supports or not the alpha encoding: select the most suitable format for the image to encode.

Disadvantages:

- No compression: the image size in bytes is proportional to the number of pixels, the transparency, and the bits-per-pixel.

Select one the following format to use a generic format:

- ARGB8888: 32 bits format, 8 bits for transparency, 8 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return c;
}
```

- RGB888: 24 bits format, 8 per color. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return c & 0xffffff;
}
```

- ARGB4444: 16 bits format, 4 bits for transparency, 4 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
   | ((c & 0xf0000000) >> 16)
   | ((c & 0x00f00000) >> 12)
   | ((c & 0x0000f000) >> 8)
   | ((c & 0x000000f0) >> 4)
   ;
}
```

- ARGB1555: 16 bits format, 1 bit for transparency, 5 per color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
   | (((c & 0xff000000) == 0xff000000) ? 0x8000 : 0)
   | ((c & 0xf80000) >> 9)
   | ((c & 0x00f800) >> 6)
   | ((c & 0x0000f8) >> 3)
   ;
}
```

- RGB565: 16 bits format, 5 or 6 per color. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
   | ((c & 0xf80000) >> 8)
   | ((c & 0x00fc00) >> 5)
   | ((c & 0x0000f8) >> 3)
   ;
}
```

- A8: 8 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0xff - (toGrayscale(c) & 0xff);
}
```

- A4: 4 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (0xff - (toGrayscale(c) & 0xff)) / 0x11;
}
```

- A2: 2 bits format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (0xff - (toGrayscale(c) & 0xff)) / 0x55;
}
```

- A1: 1 bit format, only transparency is encoded. The color to apply when drawing the image, is the current GraphicsContext color.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (0xff - (toGrayscale(c) & 0xff)) / 0xff;
}
```

- C4: 4 bits format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (toGrayscale(c) & 0xff) / 0x11;
}
```

- C2: 2 bits format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (toGrayscale(c) & 0xff) / 0x55;
}
```

- C1: 1 bit format with grayscale conversion. Image is always fully opaque.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return (toGrayscale(c) & 0xff) / 0xff;
}
```

- AC44: 4 bits for transparency, 4 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
  | ((color >> 24) & 0xf0)
  | ((toGrayscale(color) & 0xff) / 0x11)
  ;
}
```

- AC22: 2 bits for transparency, 2 bits with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
  | ((color >> 28) & 0xc0)
  | ((toGrayscale(color) & 0xff) / 0x55)
  ;
}
```

- AC11: 1 bit for transparency, 1 bit with grayscale conversion.

```
u32 convertARGB8888toRAWFormat(u32 c){
 return 0
  | ((c & 0xff000000) == 0xff000000 ? 0x2 : 0x0)
  | ((toGrayscale(color) & 0xff) / 0xff)
  ;
}
```

Figure 8.6. Generic Output Format Examples

```
image1:ARGB8888
image2:RGB565
image3:A4
```

8.3.6.2.4. RLE1 Output Format

The image engine can display embedded images that are encoded into a compressed format which encodes several consecutive pixels into one or more 16-bits words. This encoding manages a maximum alpha level of 2 (alpha level is always assumed to be 2, even if the image is not transparent).

- Several consecutive pixels have the same color (2 words).
  - First 16-bit word specifies how many consecutive pixels have the same color.
  - Second 16-bit word is the pixels' color.
- Several consecutive pixels have their own color (1 + n words).
  - First 16-bit word specifies how many consecutive pixels have their own color.
  - Next 16-bit word is the next pixel color.
- Several consecutive pixels are transparent (1 word).
  - 16-bit word specifies how many consecutive pixels are transparent.

Advantages:

- Supports 0 & 2 alpha encoding.
- Good compression when several consecutive pixels respect one of the three previous rules.

Disadvantages:

- Drawing an image is slightly slower than when using Display format.

Figure 8.7. RLE1 Output Format Example

```
image1:RLE1
```

8.3.7. Fonts

8.3.7.1. Overview

Fonts are graphical resources that can be accessed with a call to `ej.microui.display.Font.getFont()`. To be displayed, these fonts have to be converted at build-time from their source format to the display raw format by the font generator tool. Fonts that must be processed by the font generator tool are declared in MicroEJ Classpath `*.fonts.list` files. The file format is a standard Java properties file, each line representing a `/` separated resource path

relative to the MicroEJ classpath root referring to a MicroEJ font file (usually with a `.ejf` file extension). The resource may be followed by optional parameters which define :

- some ranges of characters to embed in the final raw file
- the required pixel depth for transparency.

By default, all characters available in the input font file are embedded, and the pixel depth is `1` (i.e 1 bit-per-pixel). Example:

Figure 8.8. Font Generator **`*.fonts.list`** File Example

```
# The following font is embedded with all characters
# without transparency
com/mycompany/MyFont1.ejf

# The following font is embedded with only the latin
# unicode range without transparency
com/mycompany/MyFont2.ejf:latin

# The following font is embedded with all characters
# with 2 levels of transparency
com/mycompany/MyFont2.ejf::2
```

MicroEJ font files conventionally end with the `.ejf` suffix and are created using the Font Designer (see Section 10.2, “Font Designer”).

8.3.7.2. Font Range

The first parameter is for specifying the font ranges to embed. Selecting only a specific set of characters to embed reduces the memory footprint. Several ranges can be specified, separated by `;`. There are two ways to specify a character range: the custom range and the known range.

8.3.7.2.1. Custom Range

Allows the selection of raw Unicode character ranges.

Examples:

- `myfont:0x21-0x49`: Embed all characters from 0x21 to 0x49 (included).
- `myfont:0x21-0x49,0x55`: Embed all characters from 0x21 to 0x49 and character 0x55
- `myfont:0x21-0x49;0x55`: Same as previous, but done by declaring two ranges.

8.3.7.2.2. Known Range

A known range is a range defined by the "Unicode Character Database" version 9.0.0 available on `http://www.unicode.org/`. Each range is composed of sub ranges that have a unique id.

- `myfont:basic_latin`: Embed all *Basic Latin* characters.
- `myfont:basic_latin;arabic`: Embed all *Basic Latin* characters, and all *Arabic* characters.

8.3.7.3. Transparency

The second parameter is for specifying the font transparency level (1, 2, 4 or 8).

Examples:

- `myfont:latin:4`: Embed all latin characters with 4 levels of transparency
- `myfont::2`: Embed all characters with 2 levels of transparency

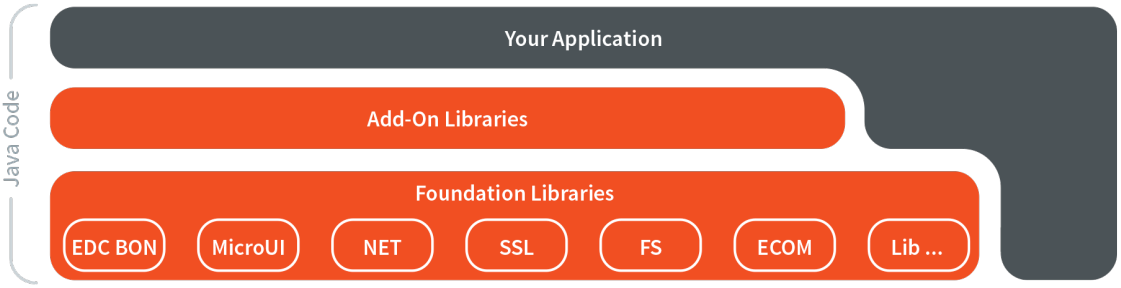
8.4. Foundation vs Add-On Libraries

A MicroEJ Foundation Library is a MicroEJ Core library that provides core runtime APIs or hardware-dependent functionality. A Foundation library is divided into an API and an implementation. A Foundation library API is composed of a name and a 2 digits version (e.g. `EDC-1.2`, `MWT-2.0`) and follows the semantic versioning (`http://semver.org`) specification. A Foundation library API only contains prototypes without code. Foundation library implementations are provided by MicroEJ Platforms. From a MicroEJ Classpath, Foundation library APIs dependencies are automatically mapped to the associated implementations provided by the platform on which the application is being executed.

A MicroEJ Add-On Library is a MicroEJ library that is implemented on top of MicroEJ Foundation Libraries (100% full Java code). A MicroEJ Add-on Library is distributed in a single JAR file, with most likely a 3 digits version and provides its associated source code.

Foundation and add-on libraries are added to MicroEJ Classpath by the application developer using Ivy (see Section 8.5, “Library Dependency Manager”).

Figure 8.9. MicroEJ Foundation and Add-On Libraries



8.5. Library Dependency Manager

MicroEJ uses Ivy (`http://ant.apache.org/ivy`) as its dependency manager for building MicroEJ classpath.

An Ivy configuration file must be provided in each MicroEJ project to solve classpath dependencies. Multiple Ivy configuration file templates are available depending on the kind of MicroEJ application created.

Example 8.5. Ivy File Template

```
<ivy-module version="2.0" xmlns:ea="http://www.easyant.org"
 xmlns:m="http://ant.apache.org/ivy/extra">
 <ivy-module version="2.0" xmlns:ea="http://www.easyant.org"
 xmlns:m="http://ant.apache.org/ivy/extra">
    <info organisation="com.mycompany" module="myapp"
 status="integration" revision="0.1.0">
        <ea:build organisation="com.is2t.easyant.buildtypes"
 module="build-application" revision="5.5.+">
         <ea:property name="test.run.includes.pattern" value="**/
_AllTests_*.class"/>
        </ea:build>
    </info>

    <configurations defaultconfmapping="default->default;provided-
>provided">
  <conf name="default" visibility="public"/>
  <conf name="provided" visibility="public"/>
  <conf name="documentation" visibility="public"/>
  <conf name="source" visibility="public"/>
  <conf name="dist" visibility="public"/>
   <conf name="test" visibility="private"/>
 </configurations>

    <publications>
    </publications>

    <dependencies>
      <!-- Declare a Foundation Library API dependency -->
      <dependency org="ej.api" name="edc" rev="[1.2.0-RC0,2.0.0-RC0["
 conf="provided->*" />
      <!-- Declare an Add-On Library dependency -->
      <dependency org="ej.library.wadapps" name="framework"
 rev="[1.2.0-RC0,2.0.0-RC0[" />
      <!-- Declare a test only library dependency -->
      <dependency org="ej.library.test" name="junit" rev="[1.0.0-
RC0,2.0.0-RC0[" conf="test->*" />
    </dependencies>
</ivy-module>
```

Dependencies are declared within the `<dependencies>` tag

- Foundation libraries are declared using the `"provided->*"` configuration. Without this, they will be considered as a regular Add-On libraries and will not be mapped to the associated implementation provided by the platform.
- Add-On libraries are declared with the default runtime configuration. All their declared dependencies will be fetched transitively.

8.6. Central Repository

The MicroEJ Central Repository is the Ivy repository maintained by MicroEJ. It contains Foundation library APIs and numerous Add-On Libraries. Foundation libraries APIs are distributed under the organization `ej.api`. All other artifacts are Add-On libraries.

For more information, please visit `https://developer.microej.com`.

Chapter 9. Virtual Device Tools

9.1. Wadapps Administration Console

The wadapps administration console is a tool that provides a command shell to manage the lifecycle of applications (install, start, stop and uninstall of an application on a device). Applications can be installed either from a local environment or directly fetched from a MicroEJ Store. The wadapps administration console has two communication variants depending on the device capabilities: either using a TCP/IP connection or a serial port.

9.1.1. Configuration

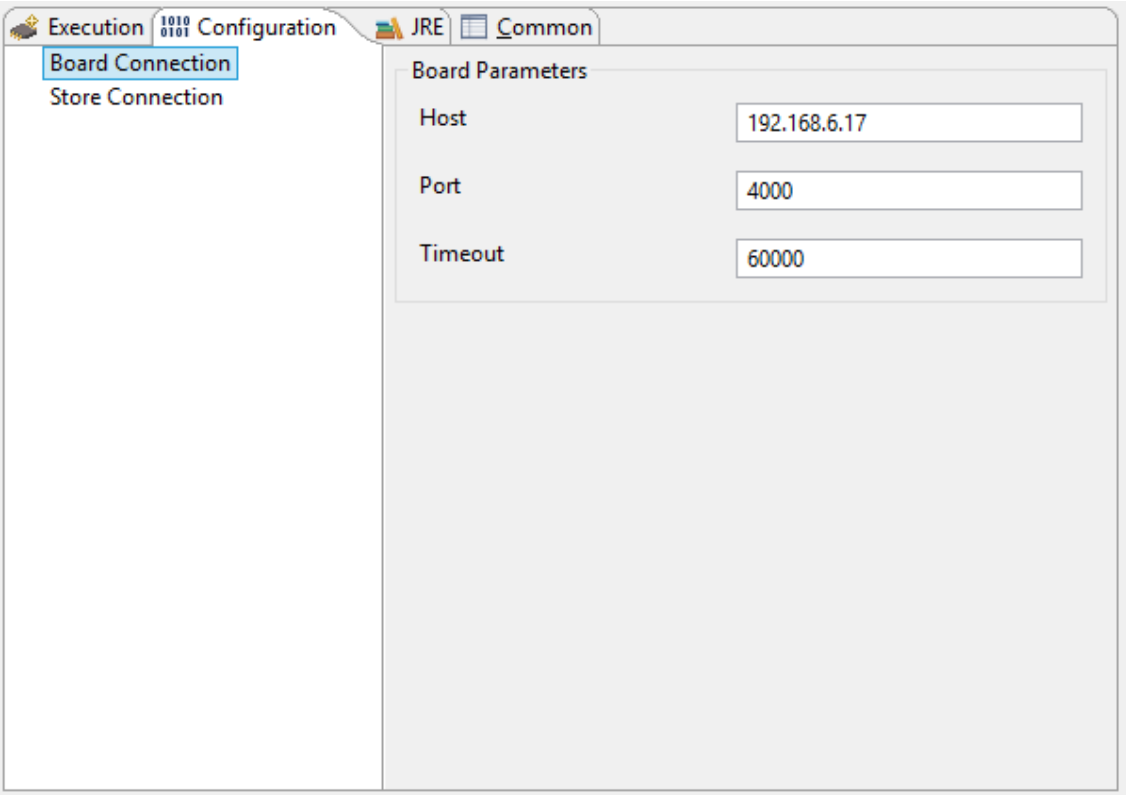
To create a new MicroEJ Tool, go to `Run > Run Configurations > MicroEJ Tool`.

In `Execution Tab`, select the virtual device that correspond to the firmware running on the device and the `Wadapps Admin Console over Socket` or `Wadapps Admin Console over Comm Port` (virtual device dependant).

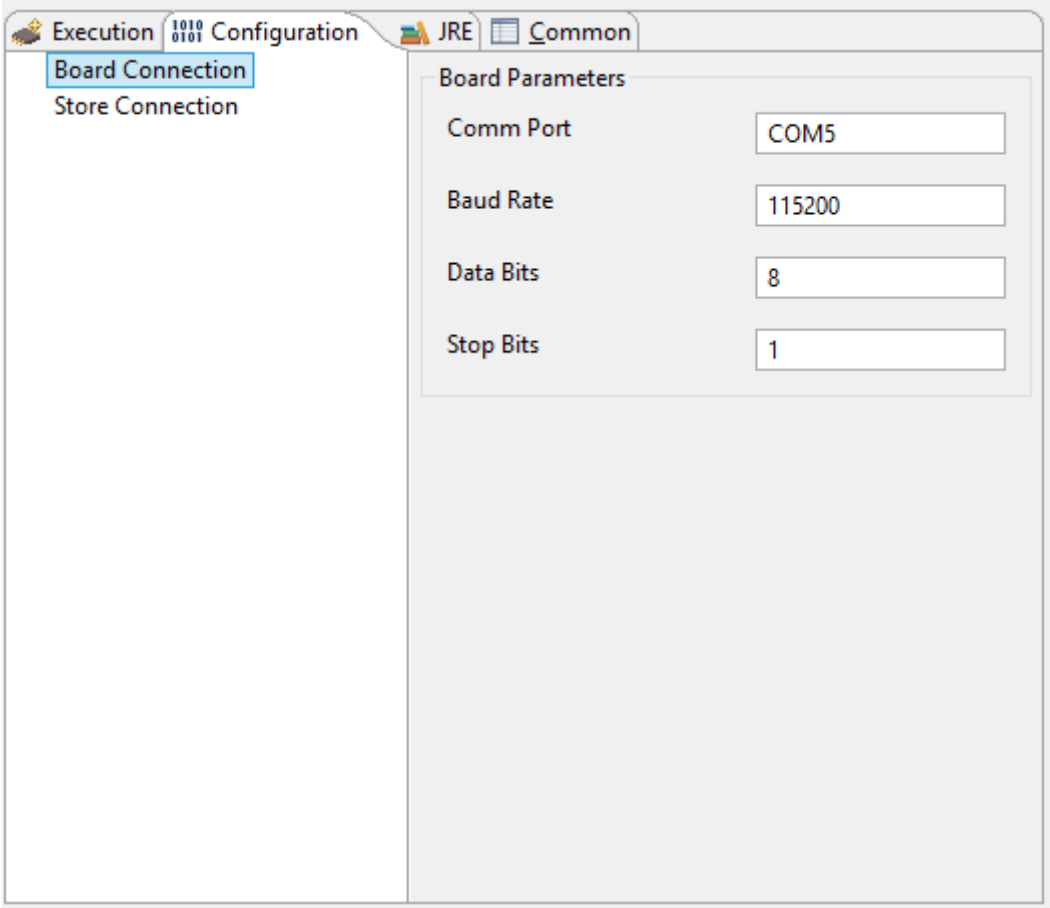
Go to the `Configuration` tab.

9.1.1.1. Board Connection Options

If the virtual device provides the `Wadapps Admin Console over Socket` variant, set the `host` with the device IP address (the device and the PC must be in the same sub network).



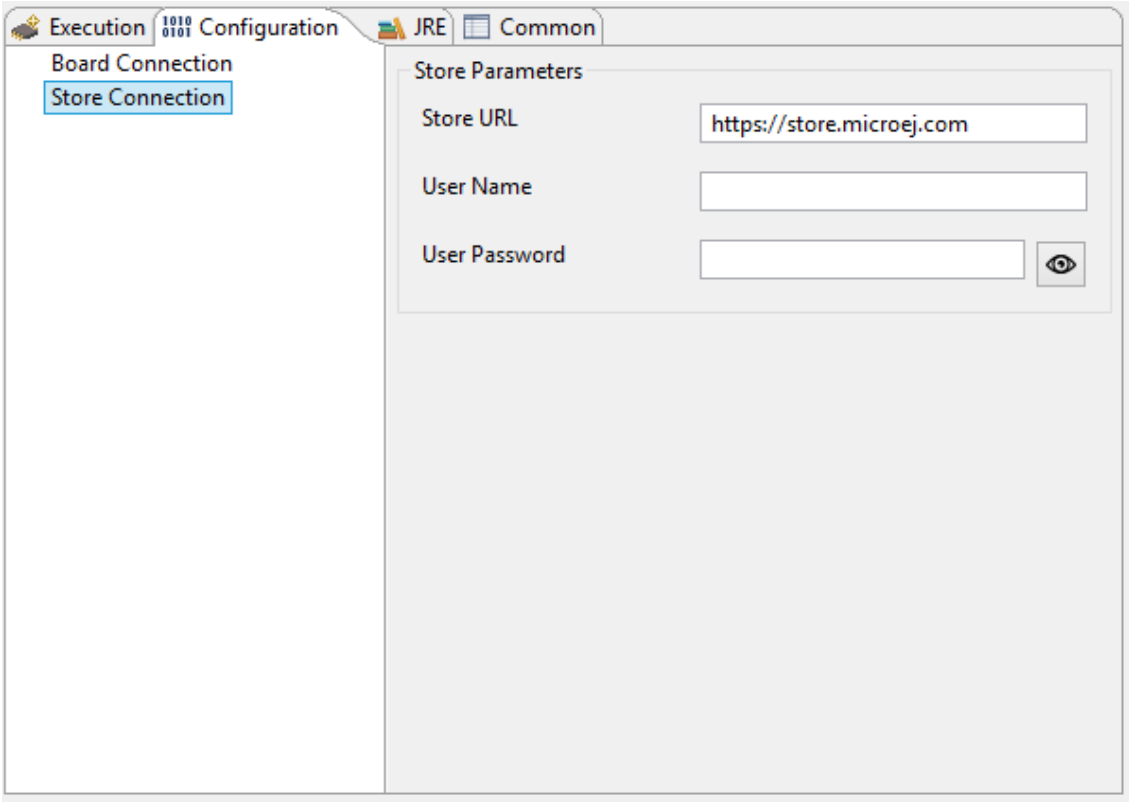
If the virtual device provides the `Wadapps Admin Console over Comm Port` variant, set the PC `Comm port` connected to the device (e.g `COMn` in windows environments and `/dev/ttyXXX` on linux environments).



Others options shall be left to their default values unless specific values are required by the firmware provider.

9.1.1.2. Store Connection Options

Set the `Store URL`, `User name` and `User Password` of the MicroEJ store. Options for connecting a MicroEJ Store are optional and can be left empty. They can be used to setup parameters `store connect` command.



9.1.2. Availables Commands

- `list`: list all applications installed on the device (id, version and current state).

```
$ list
<id1> 0.1.0 STARTED
<id2> 1.2.3 INSTALLED
```

- `start`: start an application.

```
$ start <id>
```

- `stop`: stop an application.

```
$ stop <id>
```

- `uninstall`: uninstall an application.

```
$ uninstall <id>
```

- `store`: the store command contains sub commands

- `connect`: connect to a MicroEJ Store and pairs the device (parameters are optional and can be configured through the GUI - see Section 9.1.1.2, “Store Connection Options”).

```
$ store connect <store_url> <username> <password>
```

- `install`: download and install applications from the MicroEJ Store. The application id can be retrieved in the MicroEJ Store URL when the application is selected (e.g `https://communitystore.microej.com/applications/[id]`)

```
$ store install <id1> <id2> ...
```

- `apps`: list all applications in the MicroEJ Store that can be installed.

- `install`: install an application from the local file system (`.fo` file). This is an advanced command. Most often an application is installed using the *Local Deployment* launch.

```
$ install <id> <path_to_fo>
```

For more informations about a command usage, type `help command_name`.

Chapter 10. Additional Tools

10.1. Testsuite with JUnit

MicroEJ allows to run unit tests using the standard JUnit API during the build process of a MicroEJ library or a MicroEJ application. The MicroEJ testsuite engine runs tests on a target Platform and outputs a JUnit XML report.

10.1.1. Principle

JUnit testing can be enabled when using the `microej-javalib` (MicroEJ add-on library) or the `microej-application` (MicroEJ applications) build type. JUnit test cases processing is automatically enabled when the following dependency is declared in the `module.ivy` file of the project.

```
<dependency conf="test->*" org="ej.library.test" name="junit"
 rev="[1.0.0-RC0,2.0.0-RC0["/>
```

When a new JUnit test case class is created in the `src/test/java` folder, a JUnit processor generates MicroEJ compliant classes into a specific source folder named `src-adpgenerated/junit/java`. These files are automatically managed and must not be edited manually.

10.1.2. JUnit Compliance

MicroEJ is compliant with a subset of JUnit version 4. MicroEJ JUnit processor supports the following annotations: `@After`, `@AfterClass`, `@Before`, `@BeforeClass`, `@Ignore`, `@Test`.

Each test case entry point must be declared using the `org.junit.Test` annotation (`@Test` before a method declaration). Please refer to JUnit documentation to get details on usage of other annotations.

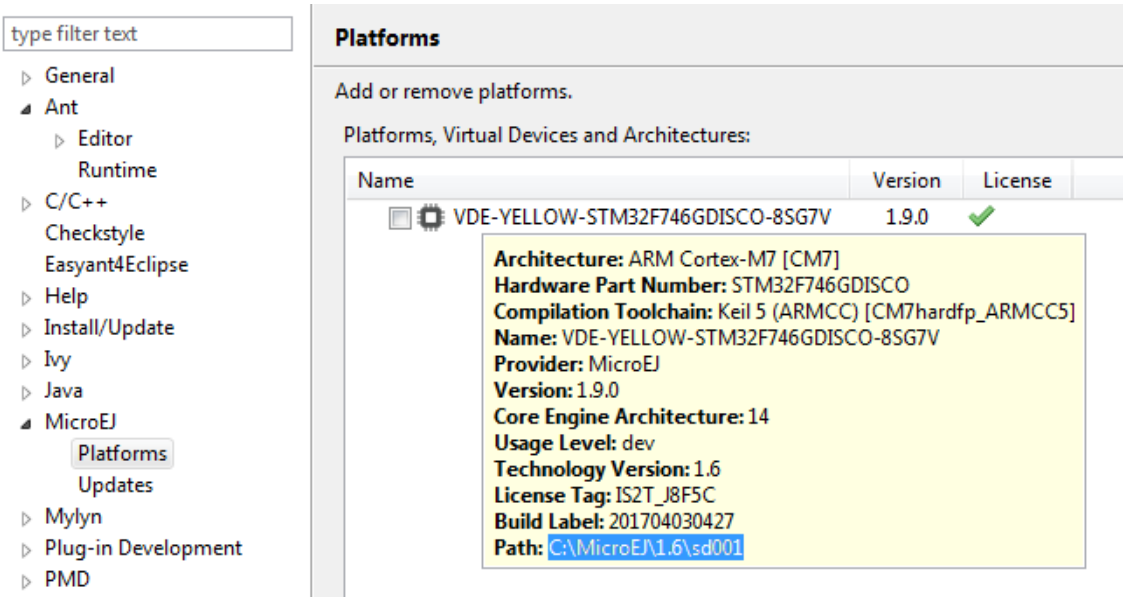
10.1.3. Setup a Platform for Tests

Before running tests, a target platform must be configured in the MicroEJ workspace. The following steps assume that a platform has been previously imported into the MicroEJ Platform repository.

Go to `Window > Preferences > MicroEJ > Platforms` and select the desired platform on which to run the tests.

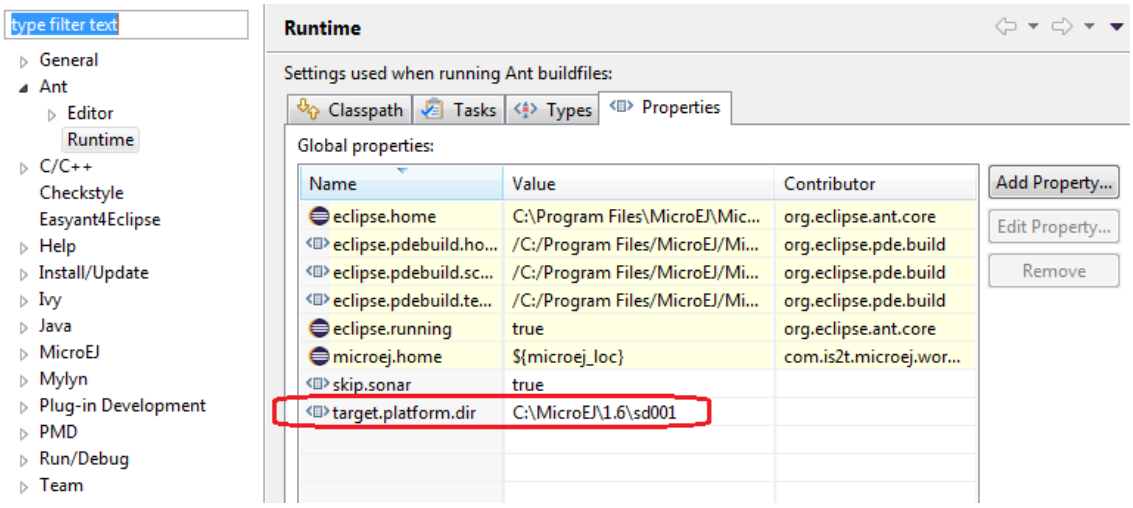
Press `F2` to expand the details.

Select the the platfom path and copy it to the clipboard.



Go to `Window > Preferences > Ant > Runtime` and select the `Properties` tab.

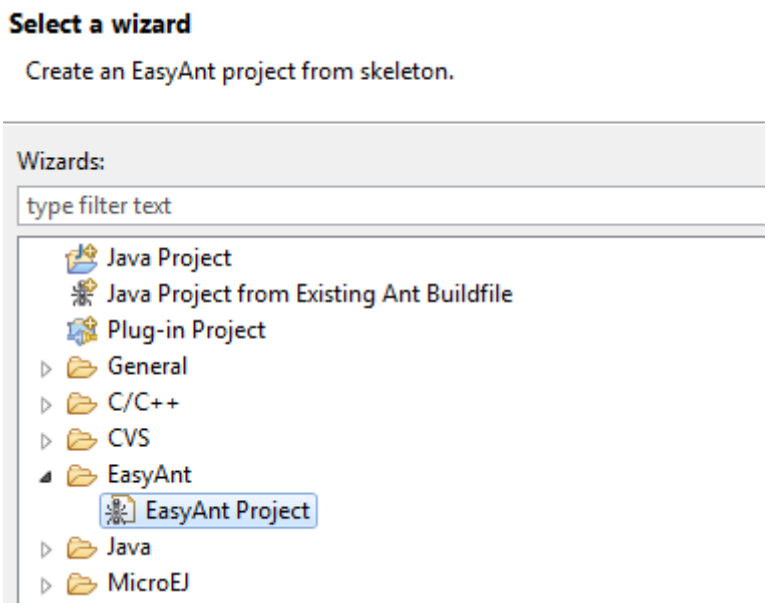
Click on `Add Property...` button and set a new property named `target.platform.dir` with the platform path pasted from the clipboard.



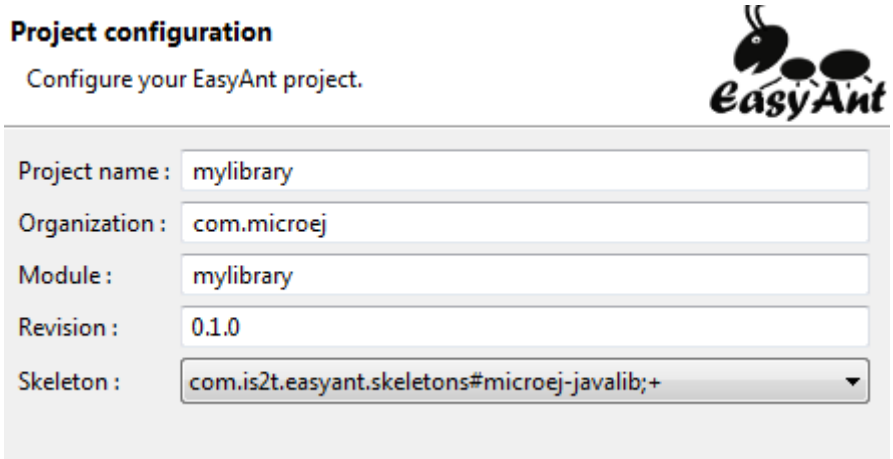
10.1.4. Setup a Project with a JUnit Test Case

This section describes how to create a new JUnit Test Case starting from a new MicroEJ library project.

Select `File > New > Project... > EasyAnt > EasyAnt Project`.



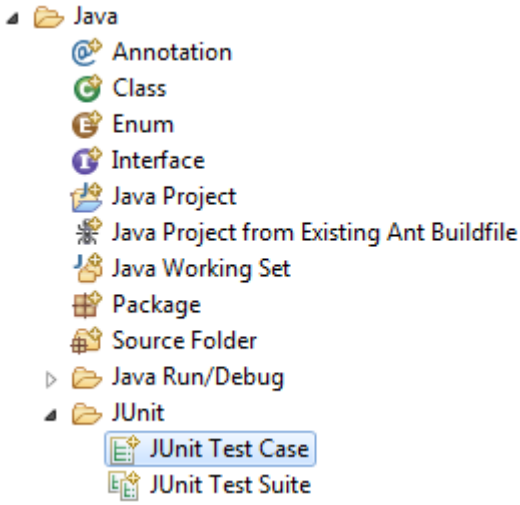
Press `Next`. Fill out project settings and select the `microej-javalib` skeleton



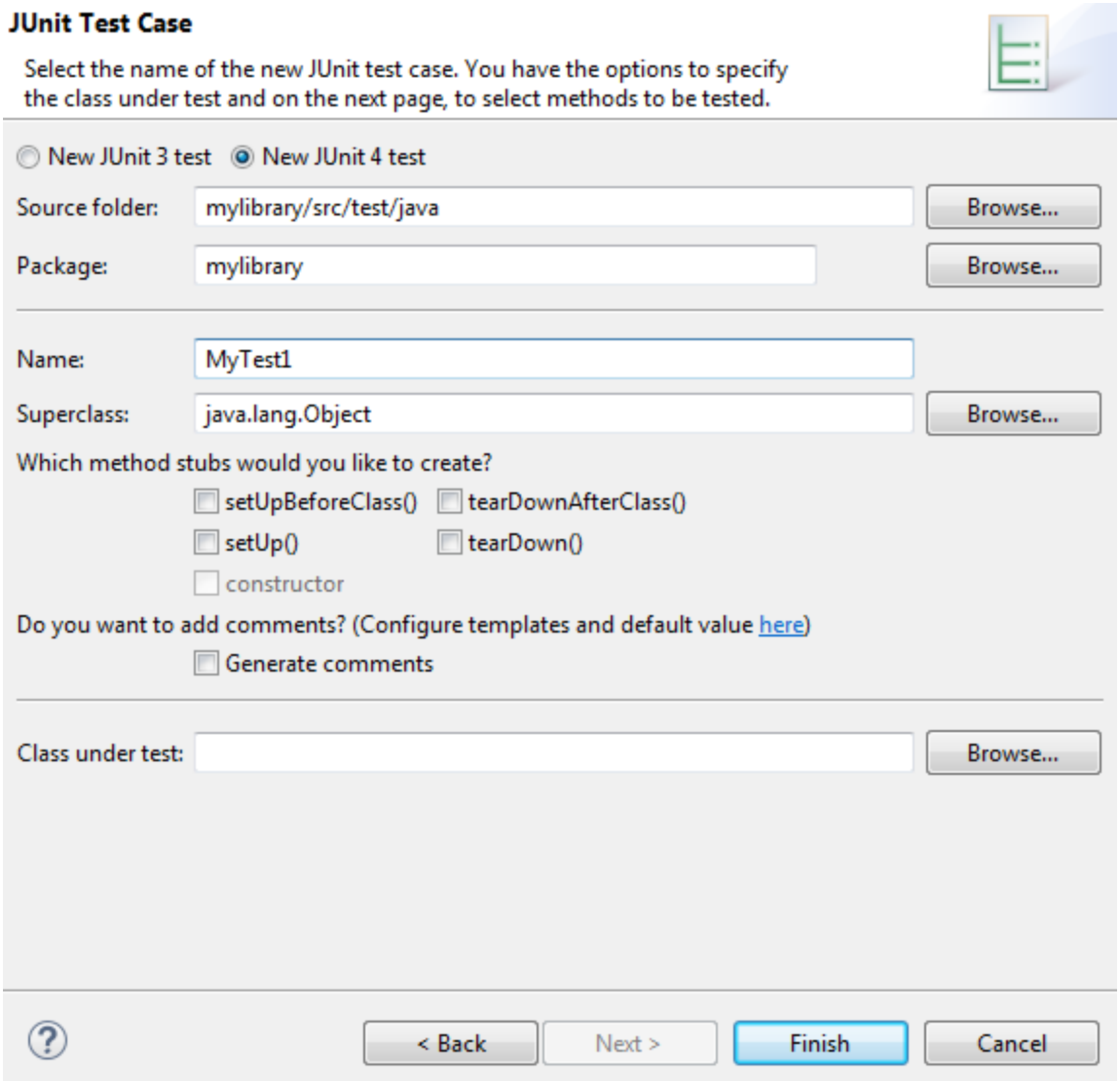
A new project named `mylibrary` is created in the workspace.

Right-click on the `src/test/java` folder and select `New > Other...` menu item.

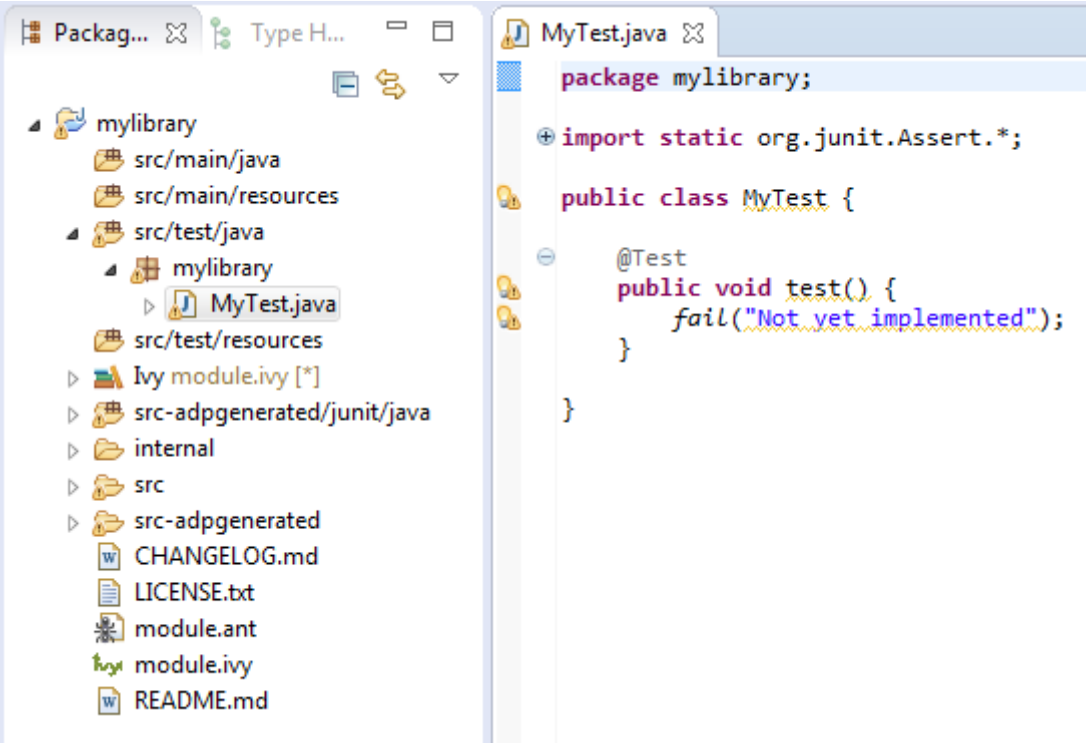
Select the `Java > JUnit > New JUnit Test Case` wizard.



Enter a test name and press `Finish`.



A new JUnit test case class is created with a default failing test case.



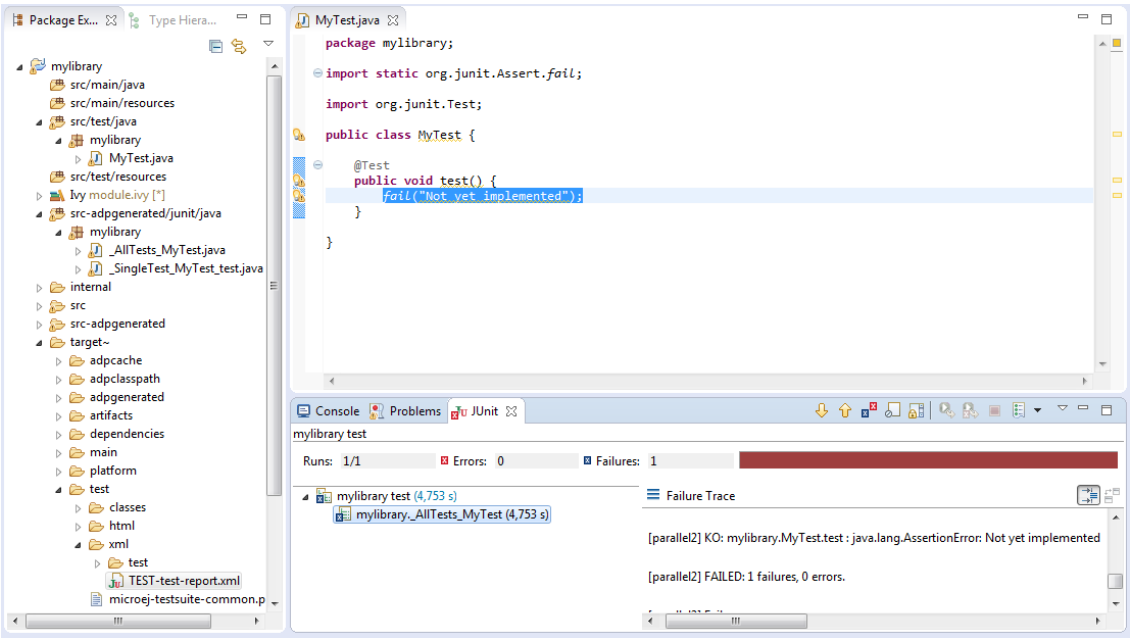
10.1.5. Build and Run a JUnit Testsuite

Right-click on the `mylibrary` project and select `Build with EasyAnt`. After the library is built, the testsuite engine launches available test cases and the build process fails in the console view.

On the `mylibrary` project, right-click and select `Refresh`.

A `target~` folder appears with intermediate build files. The JUnit report is available at `target~\test\xml\TEST-test-report.xml`.

Double-click on the file to open the JUnit testsuite report.



Modify the test case by replacing

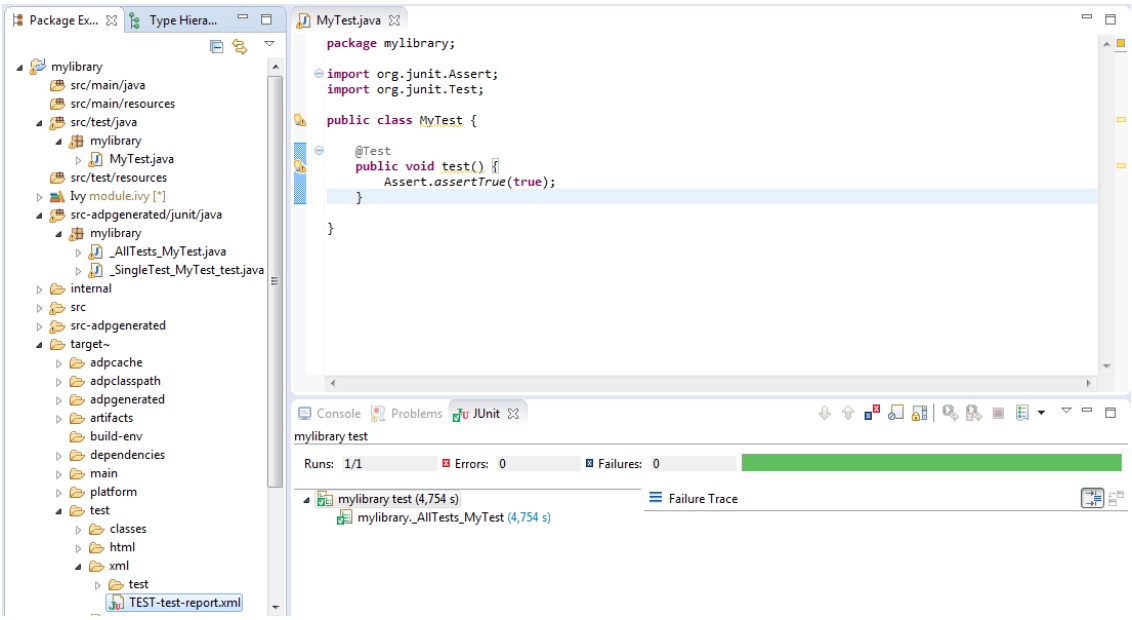
```
fail("Not yet implemented");
```

with

```
Assert.assertTrue(true);
```

Right-click again on the `mylibrary` project and select `Build with EasyAnt`. The test is now successfully executed on the target platform so the MicroEJ add-on library is fully built and published without errors.

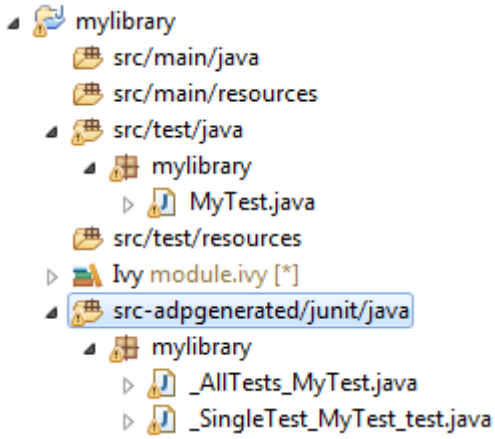
Double-click on the JUnit testsuite report to see the test has been successfully executed.



10.1.6. Advanced Configurations

10.1.6.1. Autogenerated Test Classes

The JUnit processor generates test classes into the `src-adpgenerated/junit/java` folder.



This folder contains:

- `_AllTests_[TestCase].java` files: for each JUnit test case class, a class with a main entry point that sequentially calls all declared test methods.
- `_SingleTest_[TestCase]_[TestMethod].java` files: for each test method of each JUnit test case class, a class with a main entry point that calls the test method.

10.1.6.2. JUnit Test Case to MicroEJ Test Case

The MicroEJ testsuite engine allows to select the classes that will be executed, by setting the following property in the project `module.ivy` file.

```
<ea:property name="test.run.includes.pattern" value="[MicroEJ Test
 Case Include Pattern]"/>
```

The following line consider all JUnit test methods of the same class as a single MicroEJ test case (default behaviour). If at least one JUnit test method fails, the whole test case fails in the JUnit report.

```
<ea:property name="test.run.includes.pattern" value="**/
_AllTests_*.class"/>
```

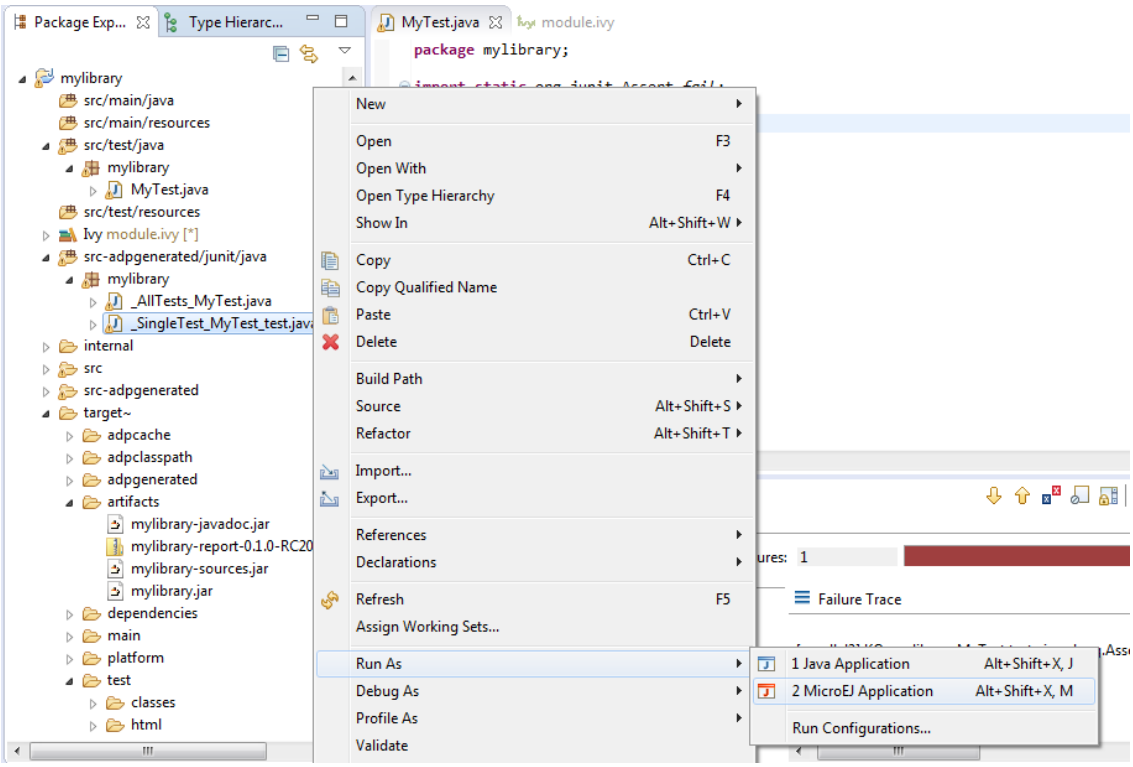
The following line consider each JUnit test method as a dedicated MicroEJ test case. Each test method is viewed independently in the JUnit report, but this may slow down the testsuite execution because a new deployment is done for each test method.

```
<ea:property name="test.run.includes.pattern" value="**/
_SingleTest_*.class"/>
```

10.1.6.3. Run a Single Test Manually

Each test can be run independently as each class contains a main entry point.

In the `src-adpgenerated/junit/java` folder, right-click on the desired autogenerated class (`_SingleTest_[TestCase]_[TestMethod].java`) and select `Run As > MicroEJ Application`.

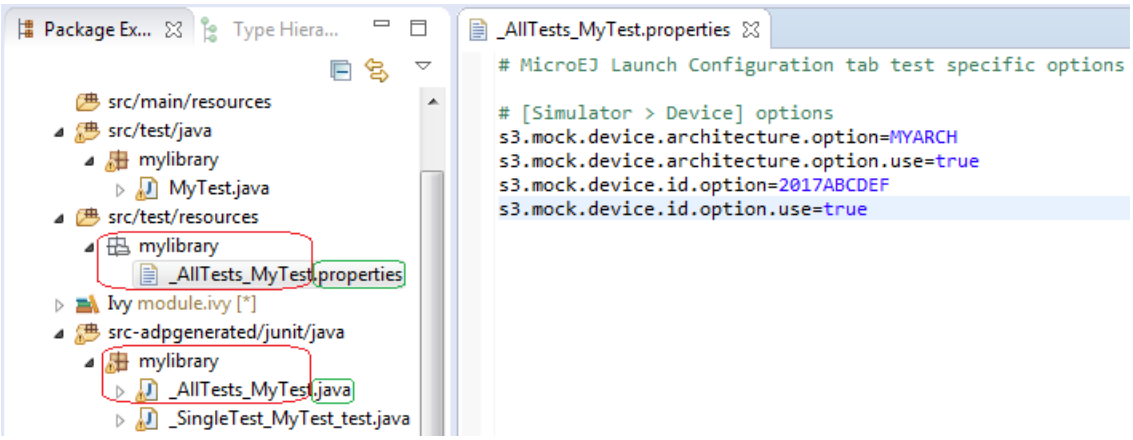


The test is executed on the selected Platform and the output result is dumped into the console.

10.1.6.4. Test Specific Options

The MicroEJ testsuite engine allows to define MicroEJ Launch options specific to each test case. This can be done by defining a file with the same name of the generated test case file with the `.properties` extension instead of the `.java` extension. The file must be put in the `src/test/resources` folder and within the same package than the test case file.

Consult the Application Launch Options Appendix of the Device Developer's Guide to get the list of available options properties.



10.2. Font Designer

MicroEJ Font Designer allows to create embedded fonts files (see Section 8.3.7, “Fonts”) from standard font files formats. The Font Designer documentation is available at: `Help > Help Contents > Font Designer User Guide`.

10.3. Stack Trace Reader

When an application is deployed on a device, stack traces dumped on standard output are not directly readable: non required types (see Section 8.3.2, “Types”) names, methods names and methods line numbers may not have been embedded to save code space. A stack trace dumped on the standard output can be decoded using the Stack Trace Reader tool.

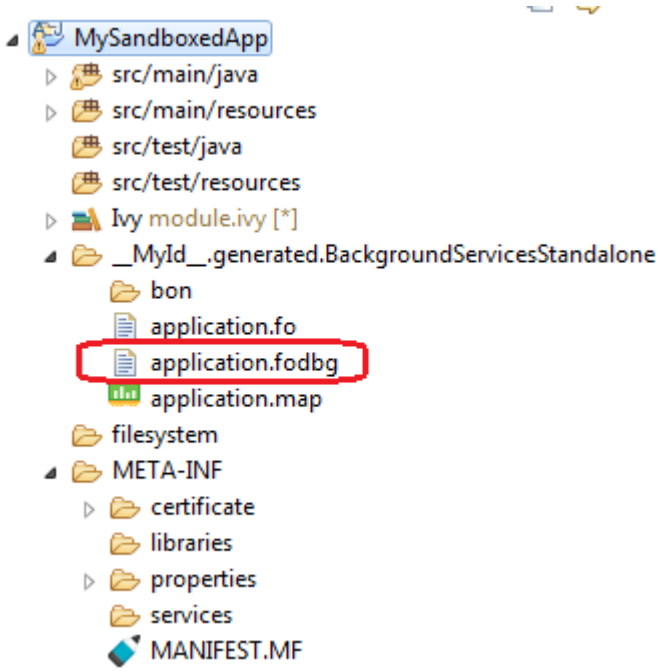
Starting from the Background Service application example (see Chapter 5, *Background Service Application*), write a new line to dump the currently executed stack trace on the standard output.

Figure 10.1. Code to Dump a Stack Trace

```
public class MyBackgroundCode implements BackgroundService {

    @Override
    public void onStart() {
        // TODO Auto-generated method stub
        System.out.println("MyBackgroundCode: Hello World");
        new Throwable().printStackTrace();
    }
```

To be able to decode an application stack trace, the stack trace reader tool requires the application binary file with debug information (`application.fodbg` in the output folder). Note that the file which is uploaded on the device is `application.fo` (stripped version without debug information).

Figure 10.2. Application Binary File with Debug Information



On successful deployment, the application is started on the device and the following trace is dumped on standard output.

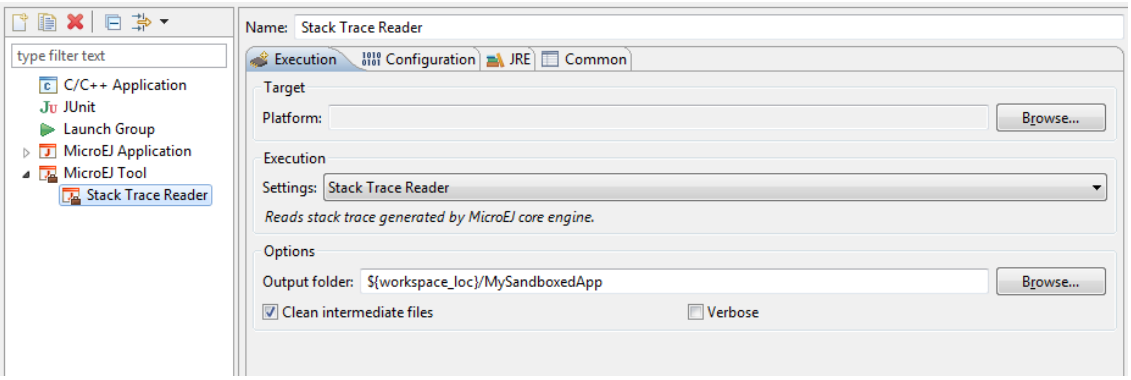
Figure 10.3. Stack Trace Output

```
MyBackgroundCode: Hello World
Exception in thread "ej.wadapps.app.default" java.lang.Throwable
     at java.lang.System.@M:0x803b334:0x803b344@
     at java.lang.Throwable.@M:0x8046aec:0x8046b02@
     at java.lang.Throwable.@M:0x805a0fc:0x805a11d@
     at appEntry.MyBackgroundCode.@F:1d48b23d5b010000d37548f1e20224d0b875cb968936fb41:0xc03800e0@@M:0xc0380bf8:0xc0380c20@
```

To create a new MicroEJ Tool configuration, right-click on the application project and click on Run `As... > Run Configurations...`.

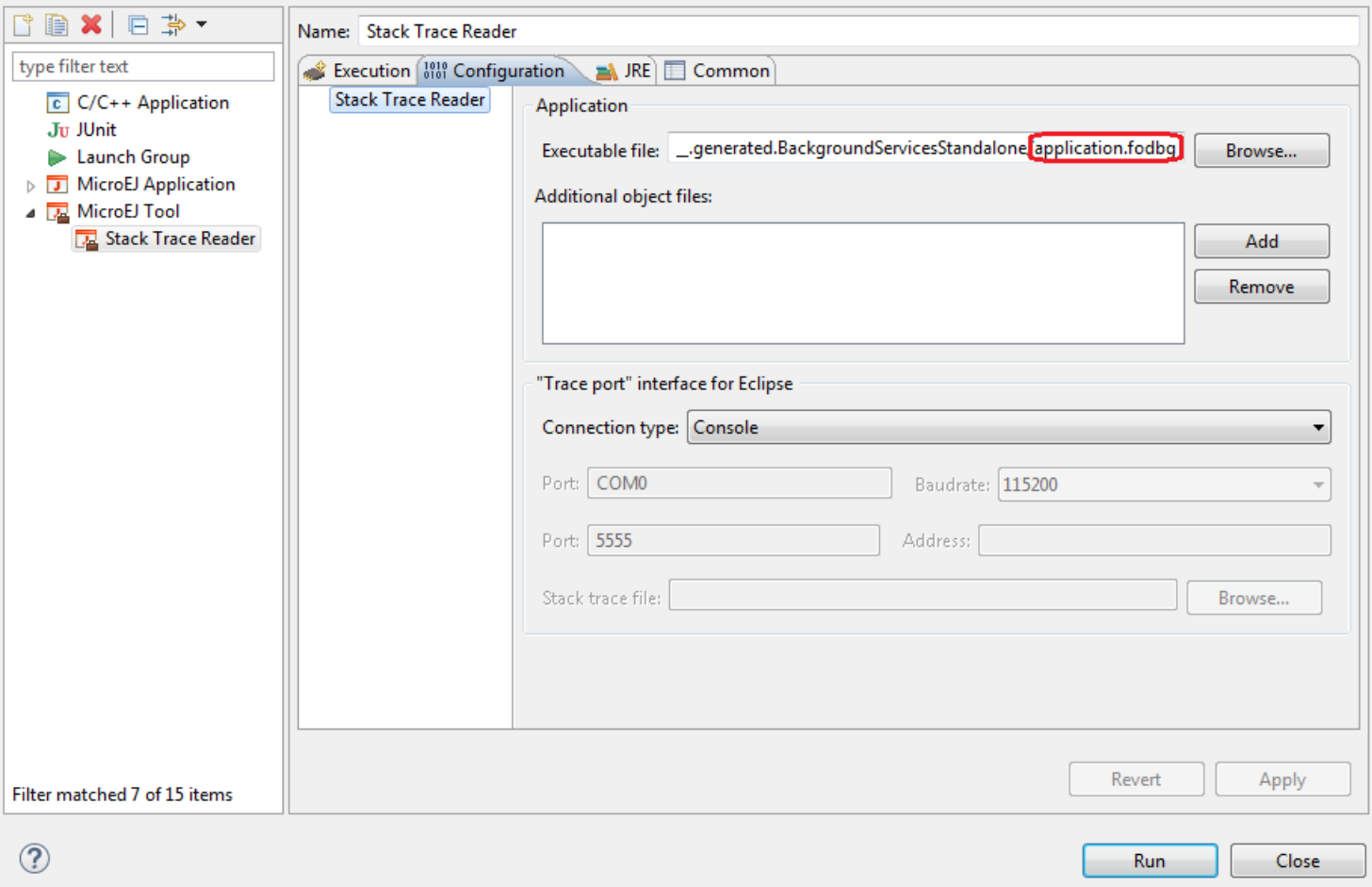
In `Execution` tab, select the `Stack Trace Reader` tool.

Figure 10.4. Select Stack Trace Reader Tool



In `Configuration` tab, browse the previously generated application binary file with debug information (`application.fodbg`)

Figure 10.5. Stack Trace Reader Tool Configuration



Click on Run button and copy/paste the trace into the Eclipse console. The decoded trace is dumped and the line corresponding to the application hook is now readable.

Figure 10.6. Read the Stack Trace

The stack trace reader can simultaneously decode heterogeneous stack traces with lines owned by different applications and the firmware. Other debug information files can be appended using the `Additional object files` option. Lines owned by the firmware can be decoded with the firwmare debug information file (optionally made available by your firmware provider).